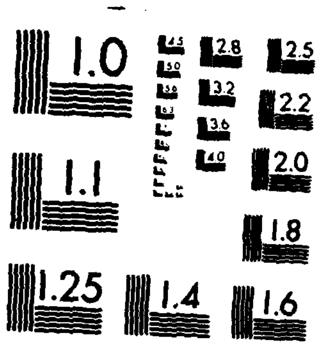
AD-A124 996 ADA* SOFTWARE DESIGN METHODS FORMULATION CASE STUDIES 1/3

REPORT(U) SOFTECH INC WALTHAM MA OCT 82

DAAK80-80-C-0187

UNCLASSIFIED F/G 9/2



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(1P.

AD A124596

ADA* SOFTWARE DESIGN METHODS FORMULATION

CASE STUDIES REPORT

OCTOBER 1982

CENTER FOR TACTICAL COMPUTER SYSTEMS
(CENTACS)

U. S. ARMY COMMUNICATIONS - ELECTRONICS COMMAND
(CECOM)

CONTRACT DAAK80-80-C-0187

PREPARED BY

SOFTECH, INC.
460 TOTTEN POND ROAD
WALTHAM, MA 02154

83 02 028 049

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO. AD-A124 996
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Ada Software Design Methods Formulation: Case Studies Report

5. TYPE OF REPORT & PERIOD COVERED

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

-

8. CONTRACT OR GRANT NUMBER(s)

DAAK80-80-C-0187

9. PERFORMING ORGANIZATION NAME AND ADDRESS

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

--

11. CONTROLLING OFFICE NAME AND ADDRESS

United States Army Communications Electronics Command, Fort Monmouth, NJ 07703

12. REPORT DATE

August, 1982

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This report is one of three in a series entitled; "ADA SOFTWARE DESIGN METHODS FORMULATION". Other reports include; "FINAL REPORT" and "APPENDICES TO FINAL REPORT".

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Case studies, design problems, embedded software systems, use and design of training simulators and equipment, general programming, design practice, programming languages, for computer programs applied to specific applications.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents a collection of case studies which illustrate in detail the effective use of Ada to solve the kinds of design problems that arise in developing embedded computer systems. Selected case studies address issues that arise in general programming and design practice and illustrate conceptual difficulties faced by novice Ada users and present examples that can be incorporated in training material. Guidelines are provided for Ada usage and style.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

SOFTECH

Section 1

INTRODUCTION

1.1 Objective

The purpose of this report is to present a collection of case studies which illustrate in detail the effective use of Ada* to solve the kinds of design problems that arise in developing embedded software systems. The case studies are of two kinds: pedagogical and observational. The pedagogical case studies present examples that can be incorporated in training material, while the observational case studies record findings that contribute to a better understanding of Ada usage issues.

1.2 Motivation

The case studies included provide concrete examples for use in Ada training materials. They illustrate

- conceptual difficulties faced by novice Ada users
- Ada solutions to design problems
- elegant uses of Ada
- Ada coding paradigms

In a different context, they provide guidelines for Ada usage and style. The selected case studies address issues that arise in embedded computer systems as well as general programming and design practice. While the code itself is drawn from embedded systems, the abstract concepts discussed concerning this code are not limited to such systems.

Ada is being considered for use throughout the system life cycle by system developers and indeed, the range of examples in this report covers different aspects of the life cycle. To explore Ada's role in the life cycle, the case studies developed in this report are organized by the major stages of the life cycle. The partitioning is discussed in Section 1.4.

The life cycle organization is also appropriate for another reason. The source material for the case studies is the work of the two contractors who participated in the Ada redesign effort (see Section 1.3). Because the examples arose during the life cycles of the two development efforts, this organization presents them in the order they occurred and provides continuity for the reader.

---

*Ada is a trademark of the Department of Defense (Ada Joint Program Office).

1.3 Approach

The Ada Software Design Methods Formulation contract was performed by the Federal Systems Group of SofTech, Inc. under contract to the Department of the Army Communications-Electronics Command (CECOM). It provided for interaction with CECOM and with two contractors, each redesigning a large-scale embedded computer system using Ada as a design and implementation language. (One system is an air defense system, the other a message switch.) Through the discussions in Technical Interchange forums which CECOM chaired, SofTech acquired the material for the case studies in this report. Questions and problems as well as solutions and ideas discussed in these meetings formed the basis for the case studies. A frequent question involved how one would express a particular paradigm or activity in Ada. Internal discussion at SofTech on the issues raised at the Technical Interchange meetings led to the selection and development of the actual case studies included in this report.

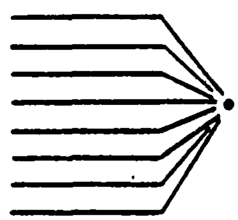
The case studies are presented in a uniform, self-contained format. Each begins with a background section which states the case study objective, describes the designer's problem, and gives a brief discussion of the problem. The detailed example then follows. It includes a problem statement, a solution outline, and a detailed solution. Finally, each case study concludes with an epilogue that discusses what was learned and how it might be applied.

1.4 Life Cycle Overview

This section is intended to give a brief overview of the system life cycle. Moreover, it outlines those phases of the life cycle which are considered in greater depth in this report.

The system life cycle may be divided into eight distinct phases, with a ninth phase which interacts with all the other ones:

- feasibility analysis
- requirements definition
- product design
- detailed design
- code and unit test
- integration
- installation
- operation and maintenance



verification and validation

The first phase of the life cycle is known as feasibility analysis. At this stage a very preliminary analysis of computing needs is done. Typical issues raised at this level question why a particular system is needed, whether the functions it proposes to implement would better be carried out by existing systems, and whether the proposed system is cost effective and economically viable. The underlying rationale, motivation and objectives for the system are established at this level.

Following this analysis, the project proceeds into the second phase, namely requirements definition. Requirements characterize the program, its behavior, its environment and the resources it will need. The inputs, outputs, and their transformation rules are identified, as well as the interfaces between the system and its environment.

The product design phase is a functional model of the system requirements. In this phase, the system architecture is established. Data items, inputs and outputs are identified, as are the functional operations on these objects. This phase enables the designer to decide which parts of the system should be implemented in software and which in hardware.

The detailed design phase and the code and unit test phase are interdependent. In detailed design the algorithms are developed to implement the functional operations defined in the previous phase. Data structures are defined for the abstract data objects identified earlier. Coding and unit testing consist of the programming and debugging needed to realize the algorithms and data structures. At the end of this phase, the pieces which will form a working version of the system exist.

The integration phase takes the pieces developed and combines them into a single working system. Interfaces between the components are tested and changes are made to the individual components in order to ensure that they interact according to specification. The hardware, developed separately from the software, is united with the software, and further testing is performed until the system is a working whole.

The installation phase refers to the installation of the system at the customer site and the customer's acceptance of it. The customer is trained in the use of this system, and thus this phase is closely linked with the operation and maintenance phase. This last phase entails the actual running of the system and its upgrading with improvements and/or fixes to bugs discovered in the course of operation.

The verification and validation phase, though identified as a separate phase, is really an integral part of each phase. It acts as a channel of communication between all the phases and promotes a better understanding between the developers responsible for the system in the different parts of the life cycle. Further, it allows cross-checking between different phases, an important activity because of the need to prove that the system does what it purports to do. Should changes in the code be found necessary in the integration phase, these changes could then be traced back to whatever level in the system development was necessary (for instance an error in the design of the system would result in an update of the design documents).

Because the scope of the two design contractors' efforts was limited to a redesign and partial recoding of an existing military system, a feasibility analysis did not form a part of their study. Given the constraints of their contracts, neither of the contractors reached the later phases of the life cycle, from the integration point on. Because their work concentrated on requirements, design, detailed design and coding, the case studies developed from this material only illustrate these four phases of the system life cycle. The remainder of the report discusses each phase in turn as well as the issues that arose at that phase. The case studies that correspond to a particular phase are placed at the end of the section devoted to that part of the life cycle (see Table 1).

Issues and observations that arose during the effort but have not been developed as case studies are documented in Appendix A as areas of future research. The appendix is also organized by life cycle phase.

TABLE 1
LIST OF CASE STUDIES

Section 2

REQUIREMENTS DEFINITION PHASE

2.1 Purpose

The requirements definition phase is critical to understanding the system to be developed. It ensures that the contractor understands the customer's needs and objectives before system design and implementation begin, thereby eliminating potential confusion and misinterpretation in a later phase of the development. In fact, both contractors noted difficulties in establishing requirements based solely on the A-Specification document. They found that this document was incomplete in parts and that information was lacking to specify system behavior under particular sets of circumstances. Clearly during this phase, all ambiguities about system requirements should be resolved so that a set of documents is generated which together constitute a precise statement of the system requirements. Subsequent sections discuss specific issues uncovered by the study.

2.2 Different Categories of System Requirements

System requirements may be divided into functional and nonfunctional requirements. Functional requirements determine system behavior. The nonfunctional kind include such factors as performance, timing, reliability and security. Questions arise as to what extent Ada can be used to express system requirements, be they functional or nonfunctional, and whether an Ada specification is sufficient.

Functional requirements lend themselves readily to expression in Ada. Classically, these are the procedures and functions which the system must accomplish. Nonfunctional requirements, however, are not so easily expressed in Ada. Ada does not have a specific construct which may be written at the beginning of a procedure to state that this operation must not take longer than say, 5 milliseconds, and that it must perform with a very high mean time between failures. As another example, Ada does not offer a formal mechanism to state the requirement that the system must use a particular military protocol and interface with certain equipment.

Several approaches may be taken in stating system requirements: they may be stated in an integrated manner, or the different categories of requirements may be stated separately. In the first method there is only one document produced, which in turn serves as the primary working document. It is the vehicle of communication between managers, hardware and software designers and engineers. One of the contractors used this approach and developed a document written in Ada. The Ada comment feature allowed the English statement of those requirements not expressible in other Ada constructs. Alternatively, the functional and nonfunctional requirements may be separated into two or more documents, an approach taken by the other contractor. The functional requirements were then expressed in Ada and the nonfunctional requirements were

expressed in some suitable form. Thus, the requirements for concurrency and timing constraints are separate from the functions to which they apply.

In addition to these requirements documents, both contractors stressed the need for graphic tools to express and reinforce the Ada or English-like requirements specification. Various tools were used, including SADT* diagrams, data flow diagrams and hierarchical function charts (system entity diagrams).

2.3 Single Threaded Protocol

It was felt by SofTech's project team that there is a "right" way and a classically "wrong" way to specify the processing required for a "logical stimulus" (for example, a message received by a message switch). The "right" approach is to separate the logical processing of a single message from

- the lower-level protocols that map logical messages into units of transmission (message blocking and deblocking, if necessary), and

- the processing steps due to the need to "multi thread" several messages.

The "wrong" approach is to mix all these concerns. For example, one might see the specification follow a single received character; one would see the character being added to a buffer, followed by a test for the buffer full condition, in which case a new buffer would be allocated and linked to the previous one, and so on -- hardly a functional specification!

It must be emphasized that the contractors followed the <u>right</u> approach; but it was recognized that the "wrong" approach <u>is</u> a classical mistake. It was suggested that somehow the single-threaded specification should be prescribed contractually as the standard way to specify certain kinds of functions.

What makes this idea particularly attractive is the analogy with the standard separation, used in the description of programming languages, between the syntax and the lexical rules; that separation gives a predefined choice of levels of abstraction and results in a standard compiler architecture.

*SADT is a trademark of SofTech, Inc.

2.4 Ada and System Requirements Definition

Both contractors used Ada in defining system requirements. There are two issues here: first, whether Ada is the appropriate language in which to state system requirements and second, whether full Ada or a subset of Ada should be used at this level. Both contractors felt that Ada could be used effectively to state requirements. Their approaches were different, however, and they illustrate the full Ada method and the subset Ada method. It should be stressed that Ada in itself was not enough to express system requirements and that additional tools were needed. These complementary tools were mostly graphical in nature. (See Sections 2.5, 4.2.)

In expressing requirements in Ada, the question arose as to how one should state performance constraints. A proposed solution was to use comments to capture these constraints. An English description of the requirements was included as Ada comments, supplementing the functional design (which was expressed using Ada packages and subprograms). A frequently discussed issue was whether one could express performance requirements directly in Ada, using constructs other than comments. Examples of such a requirement were power failure and error recovery. The case study POWER FAILURE REQUIREMENTS shows how Ada's different constructs may be used to state requirements.

Another approach to requirements definition was to limit the Ada-stated definition to functional requirements, and in turn to limit the Ada constructs which could be used in this definition to control statements, assignment statements, procedures and functions. It was stressed that the resulting document in no way formulated the design of the system, and that the subprograms did not convey some kind of system hierarchy. What was presented as procedures and functions at the requirements level did not imply that the design would implement them as procedures and functions. The designer might, for example, choose a task or a package. Again, this method was used in conjunction with a graphical technique, where the graphics captured interface information intentionally omitted from the Ada specification.

Because of system concurrency requirements, it was thought that tasks might be a good vehicle to convey requirements. Given that tasks specify a parallel thread of processing, they were initially thought to be a good vehicle to convey the fact that several parts of a system must operate in parallel. This approach was not successful. Using tasks at the requirements level encouraged thinking about what operations must be performed concurrently to meet performance requirements. But a requirements specification should concentrate on defining only the needed functionality and performance constraints. It is up to the system designers to decide whether concurrent tasks are needed to meet these requirements. Furthermore, tasks at the requirements level are too restrictive because their interfaces to the outside world are limited to two construct: entries and machine representation specifications. Packages provide much more flexibility because their interface is considerably larger in scope.

For example, one might think that a requirement for a full duplex communications link should be specified by describing separate tasks to handle data input and output. But if a system is fast enough, a single task will suffice. The essence of the requirement is that data is being input and output at certain rates, and the system must not lose any data. Saying that tasks should be used for this purpose is premature system design. It is hard enough to avoid doing design while attempting to establish requirements. Having to think about whether to use a task or not complicates an already complicated process, without providing any real benefits. On the other hand, one might expect different results if tasks are used to express the logical independence of different threads of control, as opposed to physical concurrency. This approach was not pursued by either contractor, and therefore no corresponding experience was gathered in the study.

Some nonfunctional requirements do in fact lend themselves to expression in Ada. At first glance, Ada does not appear to offer a formal mechanism to state the requirements that a system must use a particular military protocol or interface with certain equipment. Upon closer analysis, however, Ada can be used to express this kind of requirement more precisely than in natural language. The case study USE OF TYPES TO DESCRIBE HARDWARE INTERFACE REQUIREMENTS illustrates this point.

2.5 What was Used Besides Ada

As mentioned earlier, other techniques were used to complement the Ada statement of requirements. Both contractors used a data dictionary. Several different graphical techniques were used, namely system entity diagrams, SADT, and system data flow diagrams. System entity diagrams were used to express the functional hierarchy of the system. One problem that was noted with this method was that it did not capture the system interfaces. As an exercise at one of the technical interchange forums, these diagrams were reworked so as to include the interfaces, and the results are presented in the case study FUNCTIONAL DESCRIPTION OF AN AIR DEFENSE SYSTEM.

2.6 Summary

The requirements definition phase of the system life cycle is best approached from several angles. Functional and nonfunctional requirements are best recorded using a combination of language and graphical techniques. While an Ada requirements statement provides textual information about the system, the graphical outputs show relationships and interfaces among system requirements.

2.7 Case Studies

The case studies illustrating this phase are:

- Power failure requirements
- Use of types to describe hardware interface requirements
- Functional description of an air defense system

POWER FAILURE REQUIREMENTS

1. BACKGROUND

Case Study Objective

To state system requirements for power failure in Ada.

Designer's Problem

Ada does not seem to support the kind of actions necessary to recover from a power failure. How do you suspend a task, save its environment and restart it later?

Discussion

Failure recovery _is_ a difficult problem, but the main difficulty is in defining the system requirements. Task suspension, etc. are implementation details; numerous solutions can be devised within the language, and the preferred approach will depend on the specific requirements.

An attempt was therefore made to define the specific requirements first. Ada was used as the requirement language.

2. DETAILED EXAMPLE

Example Problem Statement

The general requirements had already been formulated informally as follows.

- On a power failure condition, a lamp on a remote communications processor(s) front panel will be illuminated and the power fail timer started. The remote communications subsystem will retain its status and message contents for a minimum of 24 hours.

- Upon power restoration, an automatic restart procedure will be initiated which will include:

    - Notification of the power fail condition and duration to all users - local and remote.

    - Resumption of communication services at the point of interruption.

    - Extinguish the power fail lamp on the front panel.

Solution Outline

To state the requirements in Ada means to describe an ideal system exhibiting the desired behavior.

Task suspension and activation in Ada happens normally as a consequence of the priority rules. Tasks are automatically suspended when a higher-priority task becomes eligible for execution. In the idealized system, a hypothetical task, having higher priority than any other task in the system, becomes eligible for execution when a power failure occurs. This task will prevent all other tasks from executing until power is restored, at which time execution of the other tasks will resume as usual. If power is not restored within an interval of 24 hours (possibly more, see below), then the system will become useless (it will "crash").

The power failure condition is modeled as an interrupt (entry call with high priority). The requirement that the state must be preserved for <u>at least</u> 24 hours is expressed by adding a random value to the 24-hour threshold; note that the exact duration is allowed to be different for different power failure episodes. The "crash" is modeled by an infinite loop (which renders the system useless).

In the Ada program shown below, "--> " indicates a requirement stated informally (because the details are irrelevant to the problem at hand).

<u>Detailed Solution</u>

```
--  Upon occurrence of a power failure, a task grabs the processor and
--  waits for power restoration. As soon as such task releases the
--  processor, all other tasks will be automatically resumed.

with RANDOM; --    a primitive function, assumed to return a positive
             --    value of some numeric type; no particular distribution
             --    is assumed.
with CALENDAR; use CALENDAR;
separate (...)
task body POWER_FAILURE_HANDLER is
    HOURS : constant := 60.0 * 60.0;
    TIME_OF_FAILURE : TIME;
    MAX_RETENTION_TIME, ELAPSED_TIME : DURATION;
    procedure CEASE_TO_EXIST is separate;
begin
    loop
         accept POWER_DOWN;
         --> Turn on the lamp on the remote communications panel
         TIME_OF_FAILURE := CLOCK;
         MAX_RETENTION_TIME := 24.0*HOURS + DURATION(RANDOM);
                             -- i.e., at least 24 hours, possibly more
         ELAPSED_TIME := 0.0;
         FREEZE : while ELAPSED_TIME < MAX_RETENTION_TIME
                    loop  -- N.B. : This is a busy loop.
                         select accept POWER_UP;
                              exit FREEZE;
                         else null;
                         end select;
```

```
                    ELAPSED_TIME := CLOCK - TIME_OF_FAILURE;
                    end loop;
          if ELAPSED_TIME < MAX_RETENTION_TIME then
               --> extinguish lamp (see note 1 below)
               --> notify all users
          else CEASE_TO_EXIST;
          end if;
     end loop;
end POWER_FAILURE_HANDLER;

separate (...POWER_FAILURE_HANDLER)
procedure CEASE_TO_EXIST is
begin
    loop null; end loop;
end CEASE_TO_EXIST;

-- Note 1

--  As shown, recovery from power failure is not allowed to be disrupted
--  by another power failure (i.e., there must be enough battery backup
--  to do the recovery uninterruptibly).

--  ======================================================================

--  If it is so desired, the entire phenomenon can be modeled as follows:

package POWER_FAILURE_REQUIREMENTS is
end;

with RANDOM;  -- as above
package body POWER_FAILURE_REQUIREMENTS is
    task POWER_FAILURE;                   --  Models the (uncontrollable)
                                          --   occurrence of failures
    task POWER_FAILURE HANDLER is         --  specifies the system
                                          --   requirements
         entry POWER_UP;
         entry POWER_DOWN;
    end POWER_FAILURE_HANDLER;

    procedure POWER_UP renames POWER_FAILURE_HANDLER.POWER_UP;
    procedure POWER_DOWN renames POWER_FAILURE_HANDLER.POWER_DOWN;

    task body POWER_FAILURE is
    begin
         loop -- forever
              delay DURATION (RANDOM);
              POWER_DOWN;
              delay DURATION (RANDOM);
              POWER_UP;
         end loop;
    end POWER_FAILURE;
```

```
    task body POWER_FAILURE_HANDLER is separate; -- (as before)

end POWER_FAILURE_REQUIREMENTS;

-- ======================================================================

-- Note 2

-- For the interpretation of the requirements, assume the following:

-- •    The "system" is treated as one (and only one) logical processor,
        with non-preemptive scheduling.

-- •    The POWER_FAILURE task resides outside the "system" and executes
        continuously, in parallel with the system.

-- These are not implementation considerations.  The exercise has shown
-- that if Ada is to be used in stating requirements, then certain
-- characteristics, which ordinarily would be implementation defined,
-- must be (arbitrarily) specified to make the requirements unambiguous.
```

3. EPILOGUE

With regard to power failure, the example illustrates that the stated requirements (continue from the point of suspension) cannot be taken literally. Clearly a task that was in the middle of a communication protocol cannot simply resume the protocol as if nothing had happened.

As an exercise in specification, the example illustrates how requirements can be expressed in Ada. External agents are modeled as tasks running on different processors; the characteristics exhibited by such tasks (for example, their timing behavior) define the characteristics of the external stimuli.

As an exercise in tasking, the example illustrates a general technique for preempting tasks when high-priority events occur: Have a higher priority task become eligible when the event in question has occurred. To this end, the high-priority task can either call an entry that will only be accepted when the event has occurred, or (as in the exercise) accept an entry that will only be called when the event has occurred.

The example can be augmented by the following exercises:

- Modify the solution so that if power is restored within two seconds, users are not notified, and everything goes back to normal.

- An implementor may take exception to the requirements as stated, on the grounds that no time is allowed for battery re-charge. Discuss how the specification might be modified to allow the retention period to become sensitive to parameters such as the length of the last outage or the time since the outage.

The second exercise raises not only technical questions, but also interesting items for discussion, concerning customer-contractor interactions in refining the requirements.

THIS PAGE INTENTIONALLY LEFT BLANK

USE OF TYPES TO DESCRIBE HARDWARE INTERFACE REQUIREMENTS

1. BACKGROUND

Case Study Objective

To illustrate the use of Ada in specifying system hardware interface requirements.

Designer's Problem

A system has a requirement to interface with remote sites, where the interface must be able to handle different message formats and different baud rates. Can Ada be used to specify these requirements prior to system design and prior to any hardware/software partitioning?

Discussion

A specific interface has very precise requirements. A link is either synchronous or asynchronous, and either character or bit oriented. Links transmit/receive at a specific baud rate and transmit/receive only predefined message formats. These combined requirements characterize a link. C--rrently, these requirements are met by general purpose modems where the message type and bit rate is switch selectable at the modem.

Given these considerations, Ada's typing mechanism forms the basis of an elegant solution to the designer's problem. In fact, use of Ada provides more complete and unambiguous information than is contained in the English system specification.

2. DETAILED EXAMPLE

Example Problem Statement

A communications system specification states that support shall be maintained for 13 links in pairs. For each link there shall be an active (primary) and backup (redundant) link, yielding a total of 26 links. Each link shall handle 750 or 1200 bps and shall support the predefined TADIL-B, MBDL, and ATDL-1 military protocols. Additionally, at least two spare links shall be supported. Among the 13 primary links, the following distribution must be supported: eight ATDL-1, one TADIL-B and four MBDL links.

Solution Outline

As shown in Figure 1 an early attempt at stating these requirements in Ada was to include all declarations in one package specification. This approach did not capture the fact that primary and backup links come in pairs, it did not describe the characteristics of the different protocols, and it did not reflect that at least two spare links (not exactly two) were needed. A revised presentation is shown in Figures 2 and 3.

```
package LINK_COMMUNICATIONS is
     --
     -- This section contains preliminary type declarations
     --

     type ORIENTATION_CHOICE        is (CHARACTER, BIT);
     type DATA_TRANSMISSION_CHOICE  is (ASYNCH, SYNCH);
     type BAUD_RATE_CHOICE          is (BR_750, BR_1200);
     type LINK is
          record
               ORIENTATION       : ORIENTATION_CHOICE;
               DATA_TRANSMISSION : DATA_TRANSMISSION_CHOICE;
               BAUD_RATE         : BAUD_RATE_CHOICE;
          end record;
     type CODE       is new INTEGER range 1..3;
     type LINK_ID    is private;
     type LINK_TABLE is private;

     MAX_NUMBER_LINKS : INTEGER := 13;

     type PRIMARY_LINK   is array (1..MAX_NUMBER_LINKS) of LINK;
     type REDUNDANT_LINK is array (1..MAX_NUMBER_LINKS) of LINK;
     type SPARE_LINK     is array (1..2)                of LINK;

     --
     -- This section contains subprogram declarations
     --

     procedure INITIALIZE_LINKS   ( COMPLETION_STATUS : out CODE );
     procedure ADD_A_LINK         ( NEW_LINK          : LINK_ID;
                                    COMPLETION_STATUS : out CODE );
     procedure DELETE_A_LINK      ( LINK_TO_GO        : LINK_ID;
                                    COMPLETION_STATUS : out CODE );
     procedure MODIFY_A_LINK      ( LINK_TO_MODIFY    : LINK_ID;
                                    COMPLETION_STATUS : out CODE );
     procedure OBTAIN_LINK_STATUS ( A_LINK            : LINK_ID;
                                    COMPLETION_STATUS : out CODE );

     --
     -- This section contains private type declarations
     --

private

     type LINK_ID is
          record
               LINK_TYPE : LINK;
               STATUS    : CODE;
          end record;

     type LINK_TABLE is array ( 1..28 ) of LINK_ID;

end LINK_COMMUNICATIONS;
```

Figure 1. Package LINK_COMMUNICATIONS

In this early version, type CODE is declared as follows:

```
type CODE is new INTEGER range 1..3;
```

where 1, 2 and 3 would represent busy, ready and down respectively. Notice how this declaration evolved into declarations for LINK_STATUS_CHOICES and LINK_STATUS in Figure 2. The improvement in expressing the requirements is obvious.

Additional improvements in the code of Figure 1 can be found by simply examining both sets of code.

The final solution is comprised of two packages. First, a general package LINK_COMMUNICATIONS contains the following: link characteristics coded as Ada enumeration types; a link modeled as an Ada record (due to its heterogeneous components); aggregate assignments to construct record objects which define characteristics of the specific protocols to be supported; and finally procedure and function declarations to represent those operations which must typically be performed on links.

The second package, SYSTEM_LINK_COMMUNICATIONS, states the requirements of the system specification. It is written as a generic package to allow the number of spare links to vary for each system installation. This package depends, in the Ada sense of the word, on package LINK_COMMUNICATIONS and therefore, is preceded by the appropriate context specification.

Detailed Solution

Figure 2 represents the code for the general package LINK_COMMUNICATIONS. This package contains elementary type and object declarations as well as subprogram declarations necessary to perform operations on these objects. Figure 3 represents the generic package, SYSTEM_LINK_COMMUNICATIONS, with two generic parameters: the number of spare links and their type (i.e., TADIL-B, ATDL-1, or MBDL). These parameters must be supplied at the time of instantiation. This package references Figure 1 and thus the two packages represent the entire statement of the link requirements.

Notice the various type declarations. The enumeration type LINK_TYPE allows the definition of primary and redundant links, thus supporting the creation of the array LINK_PAIR, which uses those values as indices. Particularly nice is the fact that Ada permits arrays of arrays, thus enabling the type declaration for LINK_SET.

```
package LINK_COMMUNICATIONS is
     --
     -- The following are preliminary type declarations
     -- needed to define and declare objects to represent
     -- the remote interface requirements of the system.
     --

     type ORIENTATION_CHOICE       is (CHARACTER, BIT);
     type DATA_TRANSMISSION_CHOICE is (ASYNCH, SYNCH);
     type BAUD_RATE_CHOICE         is (BR_750, BR_1200);
     type LINK is
          record
               ORIENTATION       : ORIENTATION_CHOICE;
               DATA_TRANSMISSION : DATA_TRANSMISSION_CHOICE;
               BAUD_RATE         : BAUD_RATE_CHOICE;
          end record;
     type LINK_TYPE is ( PRIMARY, REDUNDANT );
     type LINK_PAIR is array ( LINK_TYPE ) of LINK;
     type LINK_SET  is array ( POSITIVE range <> ) of LINK_PAIR;

     --
     -- The following are object declarations for the remote interface.
     --

     TADIL_B : constant LINK := ( BIT, SYNCH, BR_1200 );
     ATDL_1  : constant LINK := ( BIT, SYNCH, BR_1200 );
     MBDL    : constant LINK := ( CHARACTER, ASYNCH, BR_750 );

     --
     -- The following are object declarations for the system.  The
     -- requirements state links must appear in pairs with each link
     -- having both a primary and redundant link.
     --

     TADIL_B_PAIR : constant LINK_PAIR := ( PRIMARY | REDUNDANT =>
                                                      TADIL_B );
     ATDL_1_PAIR  : constant LINK_PAIR := ( PRIMARY | REDUNDANT =>
                                                      ATDL_1 );
     MBDL_PAIR    : constant LINK_PAIR := ( PRIMARY | REDUNDANT =>
                                                      MBDL );

     -- At least 2 spare links are required

     subtype SPARE_LINK_RQMTS is INTEGER range 2..INTEGER'LAST;
```

Figure 2. Package LINK_COMMUNICATIONS

```
--
-- The following represent operations that typically must be
-- preformed on a link.  INITIALIZE_LINKS is a procedure that will
-- initialize an internal link table.  The remaining operations
-- are written as functions so the value to be returned may be
-- assigned directly to individual components of TABLE_ENTRY in
-- package SYSTEM_LINK_COMMUNICATIONS.  These subprograms are
-- preceded by necessary type declarations to enable a status code
-- for both primary and redundant links to be maintained.
--
type LINK_STATUS_CHOICE is ( BUSY, READY, DOWN );

type LINK_STATUS is array ( LINK_TYPE ) of LINK_STATUS_CHOICE;

procedure INITIALIZE_LINKS;
function ADD        ( A_LINK : LINK_SET ) return LINK_STATUS;
function DELETE     ( A_LINK : LINK_SET ) return LINK_STATUS;
function MODIFY     ( A_LINK : LINK_SET ) return LINK_STATUS;
function GET_STATUS ( A_LINK : LINK_SET ) return LINK_STATUS;

end LINK_COMMUNICATIONS;
```

Figure 2. Package LINK_COMMUNICATIONS (Continued)

```
with LINK_COMMUNICATIONS; use LINK_COMMUNICATIONS;
generic

     NUMBER_SPARE_LINKS : SPARE_LINK_RQMTS := 2;
     SPARE_LINKS        : LINK_SET;

package SYSTEM_LINK_COMMUNICATIONS is

     --
     -- This section contains type declarations needed to create an
     -- internal table.  They are declared as private types since the
     -- implementation of these entities is not relevant to the
     -- specification
     --

     type TABLE_ENTRY is private;
     type LINK_TABLE is private;

     --
     -- The system must communicate digitally with remote sites by means
     -- of military protocols TADIL_B, ATDL_1, and MBDL.  Support must
     -- be maintained for thirteen(13) active data links plus redundant
     -- links for each interactive link.  Each link shall be able to
     -- support each protocol.
     --
     -- The required initial link distribution is as follows:
     --

     SYSTEM_LINKS : LINK_SET (1..13) := (1..8 => ATDL_1_PAIR,
                                            9 => TADIL_B_PAIR,
                                       10..13 => MBDL_PAIR);


     --
     -- Additionally at least two (2) spare links must be supported.
     -- These links are assumed to be configured in pairs.
     --

     SYSTEM_SPARE_LINKS : LINK_SET (1..NUMBER_SPARE_LINKS) :=
                                                      SPARE_LINKS;
```

Figure 3. Package SYSTEM_LINK_COMMUNICATIONS

```
     --
     -- The following object declaration provides for an internal
     -- link table.
     --

     SYSTEM_LINK_TABLE : LINK_TABLE;

     --
     -- This section contains the private type declarations.
     --

private

     type TABLE_ENTRY is
          record
               THE_LINK : LINK_PAIR;
               STATUS   : LINK_STATUS;
          end record;

     type LINK_TABLE is array (1..26 + NUMBER_SPARE_LINKS) of
                                                        TABLE_ENTRY;

end SYSTEM_LINK_COMMUNICATIONS;
```

Figure 3. Package SYSTEM_LINK_COMMUNICATIONS (Continued)

Much of the discussion of the code is included as comments within the two packages. However, the following additional comments are appropriate:

Package LINK_COMMUNICATIONS

LINK_PAIR as an array of LINKS creates the structure shown in Table 1. Access to both primary and redundant links and their respective components is permitted. The choice of an unconstrained array for LINK_SET implies the number of spare links is only known when the system design is complete or perhaps when a particular system site is installed.

TABLE 1

LINK_PAIR STRUCTURE

Index	Value
PRIMARY	ORIENTATION
	DATA_TRANSMISSION
	BAUD_RATE
REDUNDANT	ORIENTATION
	DATA_TRANSMISSION
	BAUD_RATE

The structure for LINK_SET takes the form shown in Table 2. Each element of each link, whether it be primary or redundant, is accessible to read or write.

TABLE 2

LINK SET STRUCTURE

Index	Value	
1	PRIMARY	ORIENTATION
		DATA_TRANSMISSION
		BAUD_RATE
	REDUNDANT	ORIENTATION
		DATA_TRANSMISSION
		BAUD_RATE
2	PRIMARY	ORIENTATION
		DATA_TRANSMISSION
		BAUD_RATE
	REDUNDANT	ORIENTATION
		DATA_TRANSMISSION
		BAUD_RATE
...		

Finally, defining protocol characteristics with constant declarations shows that these characteristics are fixed parts of the system requirements.

Package SYSTEM_LINK_COMMUNICATIONS

LINK_TABLE has the structure depicted in Table 3. THE_LINK is of type LINK_PAIR and has the structure identified in the table. STATUS is an array with components of type LINK_STATUS and indices of type LINK_TYPE thus providing status codes for both the primary and redundant links. Finally, N = 1 + NUMBER_SPARE_LINKS

TABLE 3

LINK_TABLE STRUCTURE

Index	Value
1	THE_LINK
	STATUS
2	THE_LINK
	STATUS
...	
N	THE_LINK
	STATUS

3. EPILOGUE

Expressing a system's remote interface requirements using Ada provides examples of generics, packages, subprograms, and Ada's typing mechanism (particularly enumeration types and how proper choice of their values enhances readability). Additionally, the code shows that Ada is able to express hardware interface requirements effectively.

It is left as an exercise for the reader to encode the two package bodies and to create a main procedure that will include the necessary instantiation for a system with three spare links, one for each of the three protocols.

The reader may additionally want to consider what modifications might be made to package LINK_COMMUNICATIONS to allow it to be referenced by additional systems and thus become a "generic" communications package to be used "off-the-shelf" by a system designer.

Finally, the current set of declarations allows a programmer to modify the detailed characteristics of any link in inappropriate ways, e.g.:

```
SYSTEM_LINKS(1)(PRIMARY).ORIENTATION := CHARACTER;
```

This leaves the first link described as (CHARACTER, SYNCH, BR_1200) which does not conform to any of the allowed link protocols. How could the LINK_COMMUNICATIONS package be modified to disallow this kind of change? *(Hint: consider the effect of making LINK a private type.)*

THIS PAGE INTENTIONALLY LEFT BLANK

FUNCTIONAL DESCRIPTION OF AN AIR DEFENSE SYSTEM

1. BACKGROUND

Objective

To examine an alternative approach to expressing system functionality.

Designer's Problem

The designers were having difficulty in explaining the functionality of the Air Defense System to the review team using their selected methodologies and documentation techniques. Could another technique be used to clarify this and thus facilitate communication between the two groups?

Discussion

Understanding a system's required functionality, i.e., requirements definition, is a critical step in the development process. Furthermore, the designers must not only understand the requirements; they must be able to describe them on paper and thus communicate them to others. This is essential to effective interaction with the customer (the government), with reviewers (e.g., QA personnel), and with implementation and maintenance staff. The Air Defense System designers had selected a methodology that did not really address requirements but started more with system level design. We developed a top-level functional description of the system using SofTech's Structured Analysis and Design Technique (SADT*) to provide a basis for further discussion of the system. This case study presents this description.

2. DETAILED EXAMPLE

Example Problem Statement

The Air Defense System consists of a network of individual Air Defense Centers, each responsible for the defense of a particular geographical segment. Centers track aircraft in the area for which they are responsible, and they communicate with other Centers and with radar units, fire units, and human Center operators.

Solution Outline

Figure 1 is a top-level functional view of the role of an Air Defense Center in the overall air defense activity, prepared using the SADT methodology. In this approach, boxes represent activities, and arrows

---

*SADT is a trademark of SofTech, Inc.

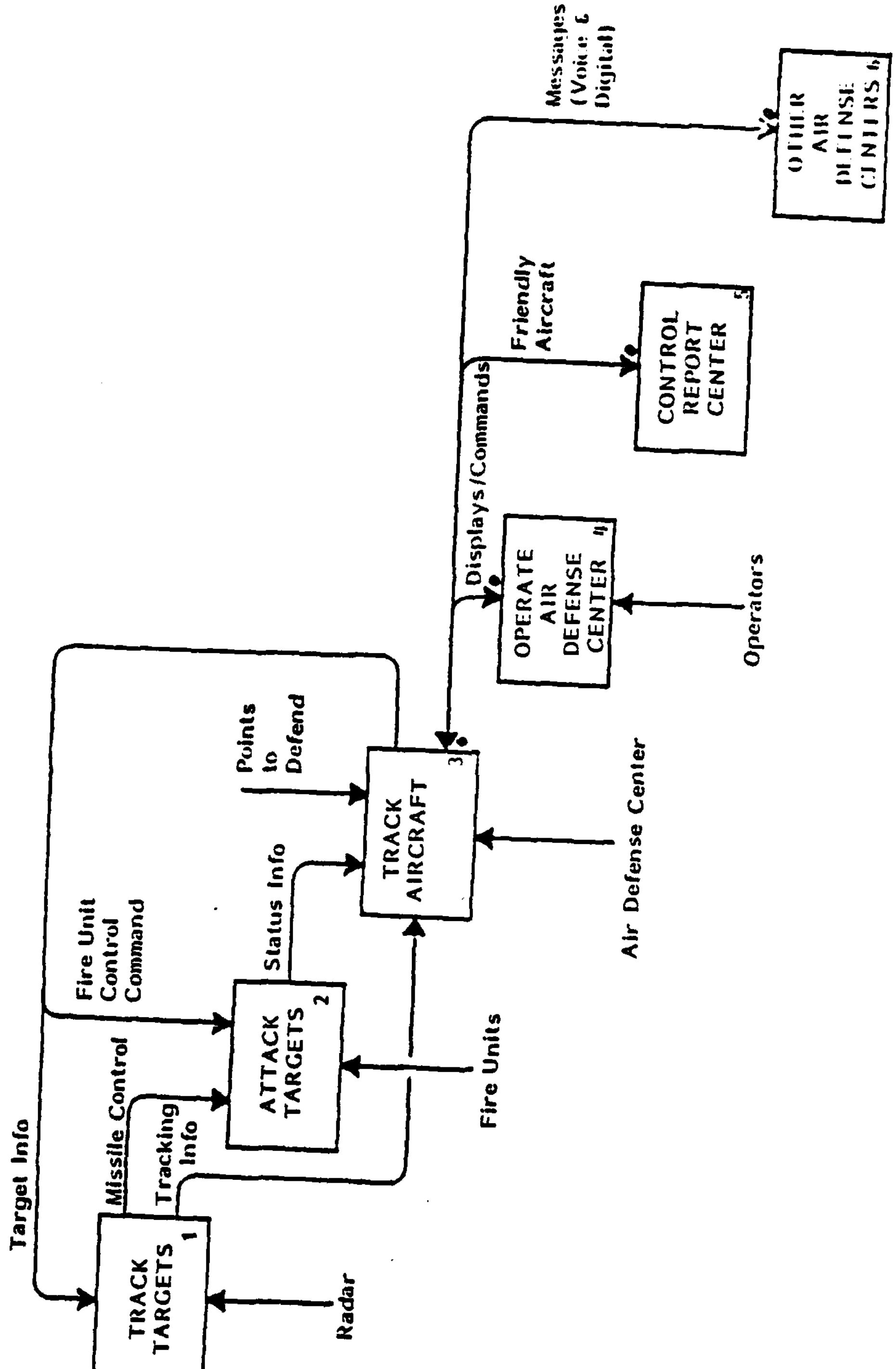


Figure 1. Defend Against Air Attack

represent data communicated among the activities. Arrows at the bottom of each activity box indicate the system entity responsible for performing that activity. Dots by the arrowheads call the reader's attention to a two way arrow.

In Figure 1, Box 3 represents the Air Defense Center. Other boxes represent the radar units, fire units, human operator report center, and other Air Defense Centers. The Air Defense Center receives tracking information from the radars and uses it to track aircraft in its assigned area. A database indicating points to defend guides this process. The Center, when appropriate, directs the fire unit to attack enemy aircraft. Interaction with the operator provides information on ongoing system activity and obtains appropriate inputs, for example in support of friend or foe determination. Information on friendly aircraft is sent to the report center. Because aircraft can move from one Air Defense Center's assigned area to another's, the Center also maintains voice and digital communication with other Centers.

Figure 2 summarizes the functionality of the Air Defense Center (i.e., Box 3 of Figure 1). It shows that the inputs to the Center are:

- radar position reports
- fire unit status information
- points to defend database
- operator commands
- incoming messages from other centers

Center outputs are:

- fire unit control commands
- target information for radar units

Detailed Solution

SADT is a technique that involves hierarchical decomposition, presenting increasingly greater levels of detail. Starting with the overall view of the Air Defense Center shown in Figure 2, we developed a functional breakdown as shown in Figure 3. Figure 3 shows the three major activities performed by the Center:

- process tracking information
- evaluate threats
- assign weapons

The tracking function processes radar inputs to maintain a database of tracking information on the various aircraft in its assigned area. It also makes a friend or foe determination for each potential target and computes target predicted position. Information on target position is then used by the threat evaluation function to prioritize targets for purposes of defensive action. This information is then passed to the weapon assignment function, which directs the fire unit to attack

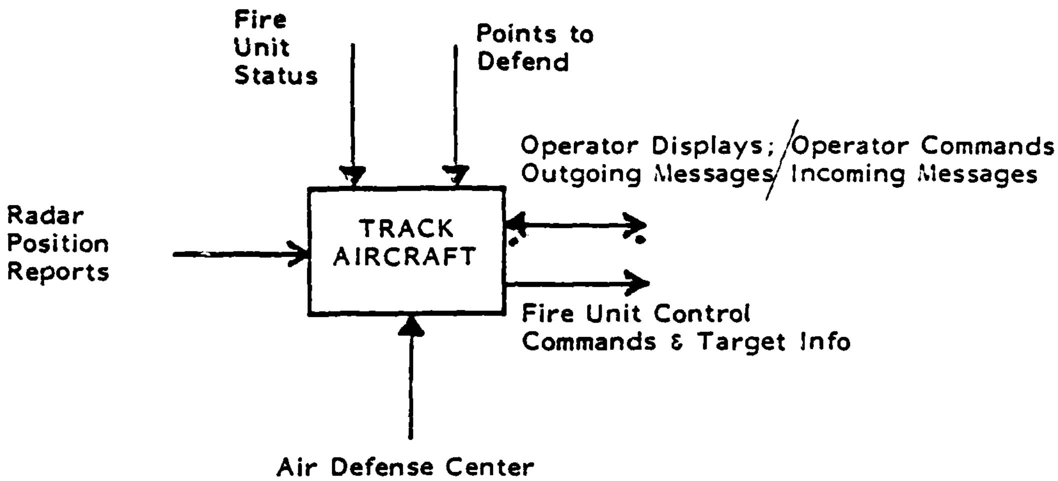


Figure 2. Track Aircraft (Summary)

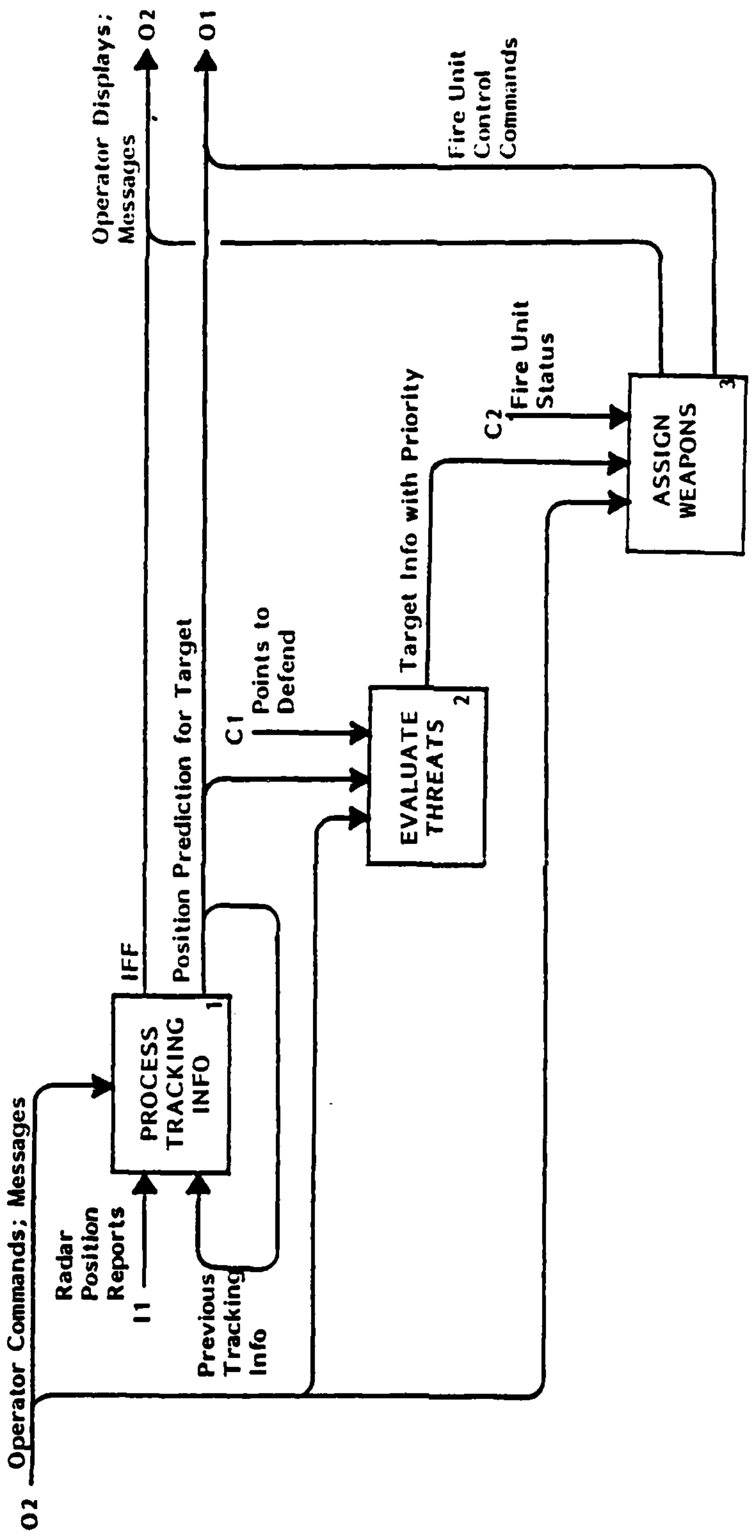


Figure 3. Track Aircraft

selected targets. The tracking function also communicates with other Centers, to hand over targets that enter their area or to obtain information on targets entering its own area.

Figure 4 then decomposes the tracking function (Box 1 of Figure 3), as this was the area selected for further analysis. This function consists of four subfunctions:

- filter jamming information
- correlate tracking information with previously known targets
- evaluate associations
- predict position

The jamming filter function obtains the most recent target input from the radar unit and filters out jamming information from the valid radar signals. It also makes a friend or foe determination based on signal content. (Here the operator's advice may be solicited.) The filtered data is then passed to the correlation function, which locates targets already being tracked with which this new input might be associated. This list of candidate targets is then passed to the association function, which selects the target that the new input is associated with and updates its track with the new input. The prediction function is then invoked to compute the predicted position for the target.

Figure 5 decomposes the prediction function (Box 4 of Figure 4). Again, this was the area selected for further consideration. The prediction function was decomposed into the following subfunctions:

- Detect Track Maneuverability
- Determine Smoothing Index
- Smooth Position
- Smooth Velocity
- Smooth Altitude
- Compute Position

The track maneuverability is analyzed as an input to determining the smoothing index for the track. The smoothing index is a set of constants selected based on the maneuverability characteristics of the object being tracked. These constants indicate anticipated variability in position, velocity, and altitude since the last reading. They are then used to compute smoothed position, velocity, and altitude values. These values are then used, together with time since the last smoothing operation, to predict a new position for the object.

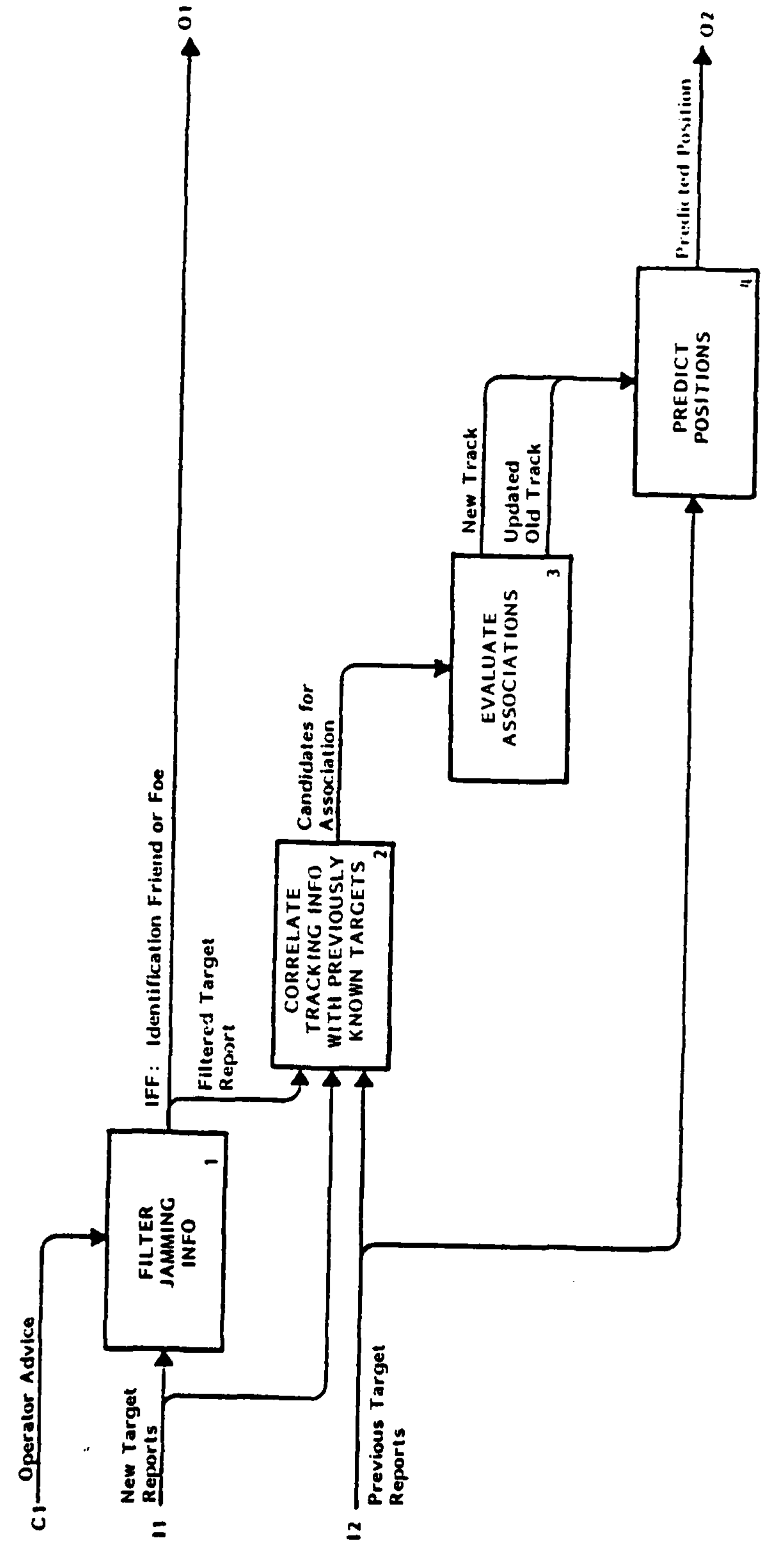


Figure 4. Process Tracking Information

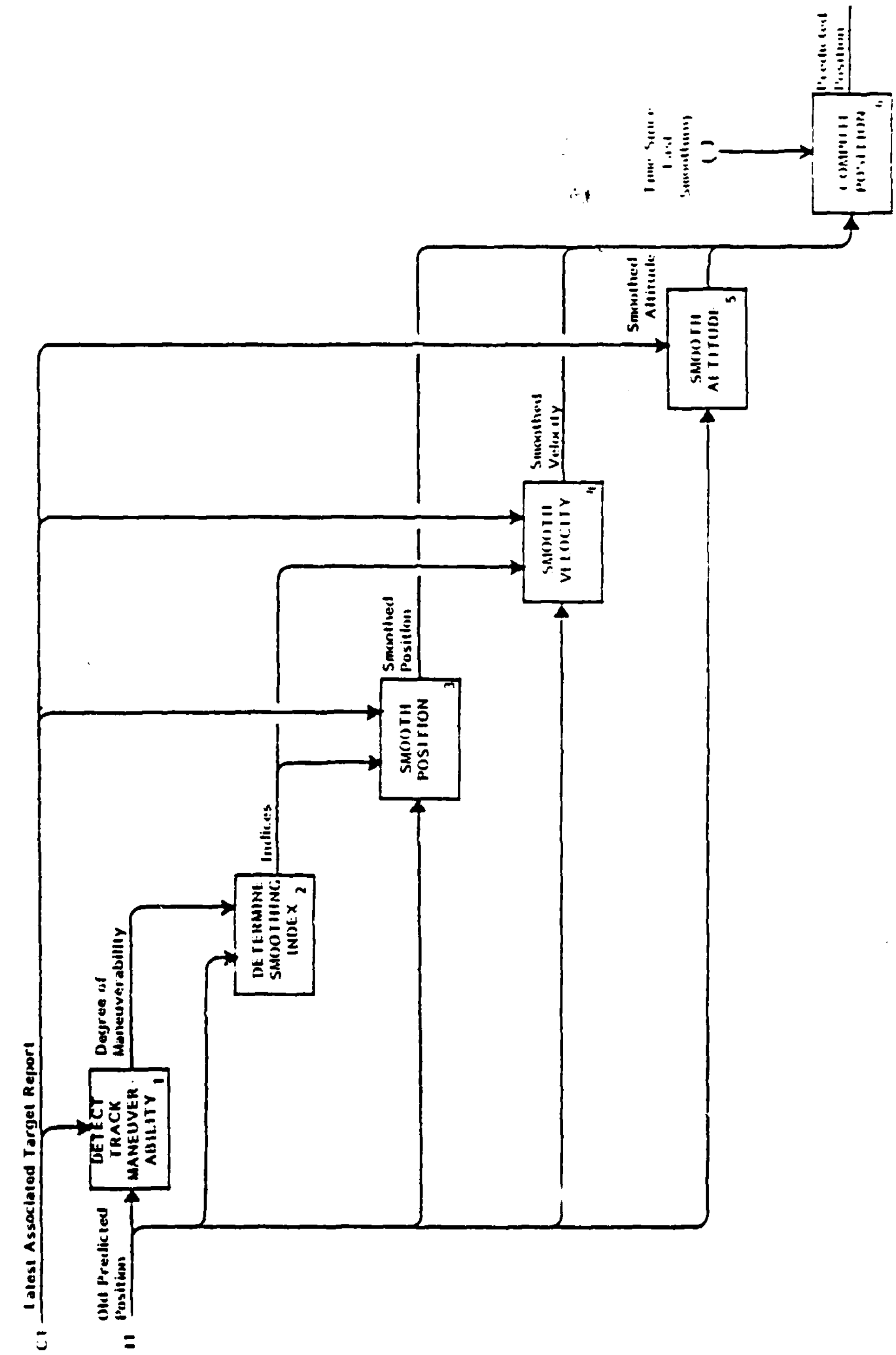


Figure 5. Predict Position

3. EPILOGUE

Requirements definition is a critical step in the system development process, and a development methodology should support this step in a way that facilitates communication with others. The SADT approach to defining the functionality of the Air Defense Center was valuable in providing a common understanding between the development team and the review team. The use of this or some similar graphic technique should be part of any overall Ada development methodology.

Section 3

PRODUCT DESIGN PHASE

3.1 Purpose

The purpose of the product design phase is to develop a system architecture which will implement the system requirements specified in the previous phase. In working with embedded systems, one of the issues that surfaces in the design phase is the development of a viable design for a real time system. This and several more general design issues are discussed in this section as they relate to Ada's tasking feature.

3.2 Real Time Systems

3.2.1 Steady State Versus Mode Transitions

A problem in designing a real time system occurs in having to design simultaneously both the system's normal operation and the procedures to follow in the event of a malfunction. From a requirements point of view, an error recovery function is independent of the steady state function. From a design point of view, however, error recovery is quite difficult in that it affects many different steady-state processes. Due to time limitations, the contractors could not pay adequate attention to the issues of system start-up and recovery, or more generally -- mode transitions.

While it would be a mistake to think of these issues as being "at a lower level," it was felt that the statement of the requirements in the "steady-state" was considerably more lucid than it would have been if mode transition considerations had been included. There is a strong suggestion that perhaps requirements definition should have two distinct phases (and correspondingly two distinct sets of documents).

In the first phase one would concentrate on the steady state behavior. This would make it easier to identify and precisely define the major functions. The resulting specification would be extremely informative, but incomplete. In the second phase one would consider explicitly the issues of control (and probably other issues such as diagnostics and fault tolerance); the result would be a complete specification, which would address these "second order" requirements.

3.2.2 Expressive Power Versus Efficiency

At least one of the contractors decided to take full advantage of the expressive power of Ada, ignoring issues of implementability at the design stage. This approach was particularly important in view of the fact that the system being designed would ordinarily be implemented with multiple/distributed processing. The implementation issues that were

deferred included low-level aspects of interprocessor communication, such as the meaning of an interprocessor rendezvous, the effect of calling a procedure that runs on a different processor, the implications of passing access types between processors, and exception propagation between different processors. This approach led to a very elegant design; when the design was complete, a number of features were identified which implementations available in the foreseable future are unlikely to support (for example, the ability to have distributed collections). In such cases, it was found easy to implement the equivalent feature in user code.

For example, where the design called for a task A to allocate a task of type T in a remote processor, it was simple to create an intermediate task in the remote processor which would receive a message from A and do the actual allocation.

Methodologically, this is an important result, since the initial design was extremely lucid and quickly derived. The manageability of the design allows the designer to define exactly the requirements of the support modules to be developed in the second stage; such modules can then be made efficient, because the exact extent of the desired generality is known.

3.3 Design Issue: Package Versus Task

As at the requirements level (see Section 2.4) the issue of packages versus tasks was an important and frequent topic of discussion at the design phase. There are two aspects to this issue.

1) What is the appropriate way to use these Ada constructs?

2) In which situations should one declare a task, and in which should one declare a package?

Tasks and packages are different orthogonal concepts. A task, like a subprogram, is a focus of control; a package is a locus of functionality. The discomfort exhibited by several designers, and the fact that the issue was strongly colored with often inappropriate performance concerns, reveal a need for better pedagogical tools and for specific paradigms and guidelines.

3.4 Data Objects and Information Hiding

In addition to designing the functions of a system, determining the data that flows through the system is a critical aspect of system design. One of the concerns in developing the data objects is information hiding. The data must be developed with sufficient levels of abstraction, to correspond to the levels of abstraction in the functional decomposition. Packages are a way of enforcing levels of abstraction. Neither contractor was entirely successful using these ideas. Both

contractors observed that, while the concepts of data abstraction and information hiding are reasonably clear, it is difficult to use the concepts effectively. It was noted, for example, that no guidelines exist for selecting an abstraction or for comparing two candidate designs based on abstraction principles. Consequently, the designers felt more comfortable with mature methods, backed by textbooks and case studies. The benefits of data abstraction were nonetheless recognized, and the concepts were applied with partial success, but more work seems to be needed in the development of a data abstraction methodology. (Reference A.2.4.)

In developing the data objects, the designer formulates type declarations. An interesting area here is the use of private types. Using private types at the design level allows implementation decisions to be deferred. Their use at the coding level has an additional effect on system security and testability. Here too, specific problems arise. For example, given a conceptual data structure, should it be declared as a record, or should it be declared as a private type, with subprograms to access the conceptual components? In the latter case, how would one document graphically the relationship of "components" to the "conceptual record"? In particular, how would the notation differ from that used for "concrete" data structures? When should a private type be limited? These are some of the questions that illustrate the discomfort experienced by novices. The questions point out areas where a methodology based on data abstraction might be "tightened" by removing some of the subjectivity and arbitrariness. (Reference A.2.5.)

There was considerable curiosity about the term "object-oriented design," a concept which originated in the late sixties and seems to be rising in popularity. Pertinent papers and books exist in the literature on languages such as CLU, Simula, Alphard, and Smalltalk, and also on the actor model of computation. Surprisingly little, however, has been written in a tutorial vein, with the possible exception of Peter Naur's "Programming by Action Clusters" (BIT #9,3, pp. 250-258, 1969).

One of the contractors experimented with this concept and attempted to consolidate the existing ideas into a formal methodology. Due to time limitations they could not complete this study. (Some of the difficulties have been discussed above.) Nevertheless, they found the exercise beneficial. In fact they subsequently applied the concepts informally but effectively.

Both functions and data are identified in the design phase, and the designer must determine the extent to which the data should be parameterized. While the operations on the data objects are specified in the functional subprograms designed, the question arises as to whether the design is the appropriate place to specify what data is global and what objects are passed as parameters. At the top levels of design, subprograms were not parameterized, thereby deferring the decision of what to parameterize until a later stage of the design. This approach makes sense in that it enables the designer to get a complete view of the

data in the system, both where and at what level of abstraction it is used. With this global understanding of the data objects, the designer can analyze them to identify potential packages. (Reference A.2.6.) Another aspect of parameterization involves functions and procedures. What is logically a function is often written as a procedure because of the need to return both a result and a status code. Guidelines are needed to determine which status parameters should be global (allowing for functions) and which should be local (procedures).

3.5 Managing a Common Storage Pool

The requirements for one of the systems included a clause to the effect that certain actions had to be taken when a certain percentage of the dynamic storage had been allocated.

Strictly speaking, this requirement cannot be implemented in "pure Ada," for two reasons.

1. In the presence of multiple collections, the language does not specify whether a single heap is used or the heap is partitioned into "zones" corresponding to the various collections. In the second case, the meaning of the requirement becomes unclear: what should one do when one of the "zones" is full or almost full? It might well be the case that the sum of the available storage is still below the threshold at that time.

2. The programmer has no straightforward way of knowing how much storage is available. Monitoring the use of one collection is relatively simple, although unpleasant: one can encapsulate the access type into a module, and keep reference counts and the like. Attempting to monitor all collections in a central user's module will decidedly have adverse effects on the modularity of the system.

It was felt that in a real-life situation one would have to resort partially to knowledge of the implementation (to settle question 1), and possibly discuss the problem with the customer. Alternatively, one might be in a position in which the implementation can be specified to support a specific storage management policy and to provide appropriate attributes of library functions (see also Section 6.4, Customized Run-time Systems).

3.6 Tasking as a Design Tool

Tasks are an elegant design tool, in part because the designer has the flexibility to decide what unit of processing the task will represent. In using tasks for system design, there are situations where the designer has a choice between allocating one task per stimulus and one task per sensor. For example, in a radar application the designer may represent each blip as a task (stimulus) or each radar sector as a task (sensor). This particular issue is explored in more depth in the

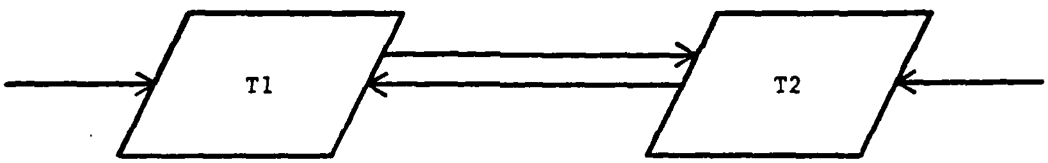
case study TASK STRUCTURE FOR A TARGET TRACKING SYSTEM. Each application will have a different set of sensors and stimuli, and in the general case, guidelines are needed which will aid the designer to select the appropriate tasking models.

In using tasking in the system design, the designer must be aware of the possibility of deadlock or deadly embrace. This occurs when two tasks are calling an entry in each other but are expecting a different entry call, as illustrated below:

```
task T1 is
     entry T1_E1;
     entry T1_E2;
end T1;

task body T1 is
     loop -- forever
          accept T1_E1;
          T2.T2_E1;
          accept T1_E2;
          T2.T2_E2;
     end loop;
end T1;
```

```
task T2 is
     entry T2_E1;
     entry T2_E2;
end T2;

task body T2 is
     loop -- forever
          T1.T1_E2;
          accept T2_E1;
          T1.T1_E1;
          accept T2_E2;
     end loop;
end T2;
```

The structure chart is shown below (see Section 4.2 for a discussion of this notation):



Task T2 could be calling entry T1_E1 while task T1 could be calling entry T2_E2. Task T1, however, is waiting to accept an entry call to T1_E2, and task T2 is waiting to accept an entry call to T2_E1. In this situation, and assuming that there are no other processes which would call these entries at this particular point in time, the processes are caught in a deadlock situation. Whenever entry calls in two or more tasks are bidirectional, this danger exists.

3.7 Modeling Finite-State Machines

Ada tasks seem to lend themselves particularly well to the implementation of processing which is naturally specified by transition diagrams. The reasons for, and limitations of, this match are an interesting area of investigation.

The reason for the match is probably the following. Consider for simplicity a finite state automaton with the terminal alphabet (a,b); any given state of the automaton can be modeled by a selective wait having the form

```
select
     accept NEXT_INPUT ('a') do ..;
or
     accept NEXT_INPUT ('b') do ..;
end select;
```

where the overall task which implements the automaton has been specified as follows:

```
task AUTOMATON is
     entry NEXT_INPUT (CHARACTER range 'a' .. 'b');
end AUTOMATON;
```

What creates the match is a combination of two factors

- The ability to accept (in the Ada technical sense) any of a number of inputs and then act accordingly.

- The blocking property of the Ada rendezvous: the automaton simply waits for an input to occur. In the task that implements the automaton one will not see any extraneous bookkeeping.

Clearly, one state can be modeled by a selective wait; alternatively, it could be a simple accept statement or entry call in which the actual input is transmitted as an actual parameter. In this second alternative, the automaton could be specified as follows:

```
task AUTOMATON is
     entry NEXT_INPUT (C : in CHAR_A_OR_B);
end AUTOMATON;
```

(where CHAR_A_OR_B is an appropriately defined subtype), and each state could have the form

```
accept NEXT_INPUT (C : in CHAR_A_OR_B);
case C is
     when 'a' => ...
     when 'b' =>
end case;
```

Clearly there are several ways to implement a state, but how should the entire automaton be implemented? The straightforward approach is to implement each state independently, and each state transition as a goto (we will call this the "verbatim" implementation). This will in general lead to unstructured code, but that is not necessarily a problem. If the

specification is given in terms of a state transition diagram and the code reflects that structure exactly, the code might well prove easier to maintain than a "structured" version which does not correspond to the specification. It must be remembered that these are abstract considerations; from a practical standpoint, the "verbatim" implementation of the state diagram may cause excessive duplication of code. In such a situation one has to decide to what extent (and how) the code should be restructured, given that the pure "verbatim" approach cannot be followed anyway for other reasons. This issue is further addressed in Section 5.3, Coding Paradigms.

A second approach is to encode state information into the guards of the selective wait. For example, suppose that our simplified automaton has two states, and that processing of the character 'b' is independent of the state. The structure of the task that implements the automaton could be the following (assuming some appropriate declarations):

```
loop
     select
          when STATE = S1 =>
               accept NEXT_INPUT ('a') do ...;
     or
          when STATE = S2 =>
               accept NEXT_INPUT ('a') do ...;
     or
          when STATE = S1 or STATE = S2 =>
               accept NEXT_INPUT ('b') do ...;
     end select;
end loop;
```

Obviously, state transitions could be effected by assigning to the variable STATE. In this approach, the code is structured around the actions, rather than the states; commonality of actions is explicitly shown, but the state transition logic is slightly less lucid. A complication that might arise in practice is that if the number of states is large, some of the guards may become unduly complex.

3.8 Alternative Task Selection

The following problem presented itself repeatedly. Let T be a task; let COND be some condition which is evaluated only once, when T is first activated, and which never changes again. Let T have the following logic:

```
if COND then
     -- allocate a task of type T1
else
     -- allocate a task of type T2
end if;
```

After allocating the tasks, then there is generally a loop which uses the allocated task:

```
loop
     ...
     if COND then
          -- call the T1 child
     else
          -- call the T2 child
     end if;
     ...
end loop;
```

The evaluation of COND inside the loop is conceptually unnecessary and misleading, since the result cannot change for different iterations. How can it be avoided? The repeated evaluation can be avoided by having the parent accept a call from the child.

This oversimplified example illustrates a general pattern which deserves further attention: Given the asymmetry of Ada rendezvous, the direction of the call can have a profound impact on the clarity and flexibility of the system. Specific guidelines for a variety of situations should be developed.

3.9 Classical Versus Ada Design Approach

An interesting outcome of the Ada redesign contracts is that certain design approaches are strongly more "Ada-like" than their classical counterpart. An instance of this distinction is seen in the fact that the classical approach does not allow for hardware design per se. In other words, hardware would not be expressed in the system design language (SDL). This was in part due to the fact that the hardware/software partitioning was done at the beginning of the system development (the requirements stage) so that by the design stage there was no need to express in SDL functions that might be implemented by hardware or by software. Given that the contractors stressed in their methodologies that the partitioning would be done after the system design phase, the need arose to express potential hardware functions in SDL. Thus, the question was raised as to whether hardware functionality can be expressed in Ada. In a technical interchange forum, such a case occurred, where hardware was successfully designed in Ada. This example has been further developed in the case study UART: EXPRESSING HARDWARE DESIGN IN ADA.

3.10 Summary

This section discussed the various ways in which Ada can be used at the design phase of the system life cycle. The Ada tasking feature was discussed in particular, because of its versatility and applicability to different kinds of design problems. This phase develops the system to the point where the resulting documents contain enough information to do a hardware/software tradeoff evaluation.

3.11 Case Studies

The case studies illustrating this phase are:

- Task structuring for a target tracking system
- UART: expressing hardware design in Ada

THIS PAGE INTENTIONALLY LEFT BLANK

TASK STRUCTURE FOR A TARGET TRACKING SYSTEM

1. BACKGROUND

Case Study Objective

The primary objective of this case study is to illustrate an approach to the selection of a tasking structure for an embedded computer program. Secondary objectives are to illustrate aspects of Ada tasking.

Designers Problem

The specific problem discussed here is the selection of a tasking structure (i.e., the identification of Ada tasks and their pattern of interaction) for the target tracking aspects of an air defense system. The target tracking systems are illustrated in Figure 1. The functions of the system are shown in Figure 2.

As shown in Figure 1(a) the air defense tracking radar conducts a 360 degree scan of the sky at a fixed rate. As it passes the north direction a synchronizing "north pulse" is passed to the target tracking system. During each scan the radar may detect reflections from a number of targets. Each reflection, termed a blip, is reported to the target tracking system. The report includes the range and bearing to the target and the range rate of change. The primary functions of the target tracking system are to correlate blips received on successive scans into a track representing the path of a single target object and to predict subsequent positions of the tracked targets as shown in Figure 1(b). The main difficulty faced by the target tracking system is that there may be a large number of targets, so it is nontrivial to associate a new blip with an existing track. Also the targets may be executing evasive maneuvers to avoid being tracked.

The functions of the target tracking system are illustrated in Figure 2. The first function "Associate Blip with Track" attempts to match new blips with existing tracks based on the predicted position of the next blip. If a match is found, the parameters describing the track are updated to reflect the new information by function 2, "Smooth Track." If no association is made the blip is taken as the beginning of a new track.

Once during each scan, independently of whether or not a new blip was received, a new predicted position is calculated for each track by function 3, "Predict Target Positions." If no new data is received for a given track within a specified number of scans, that track is discontinued.

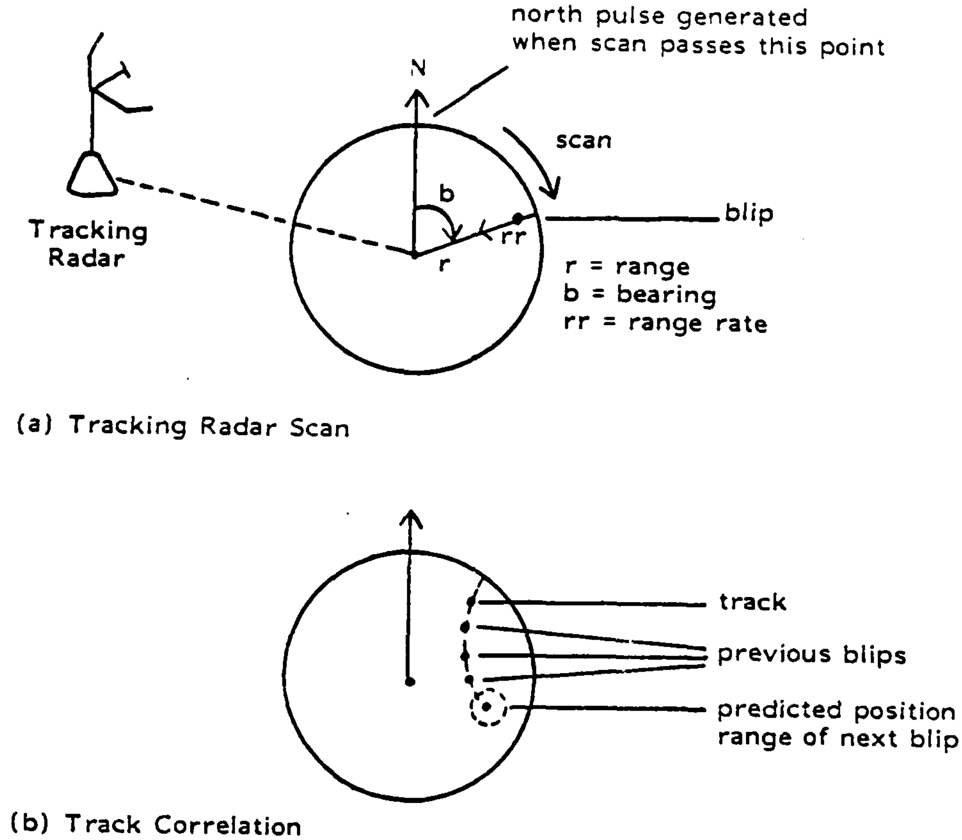


Figure 1. Target Tracking

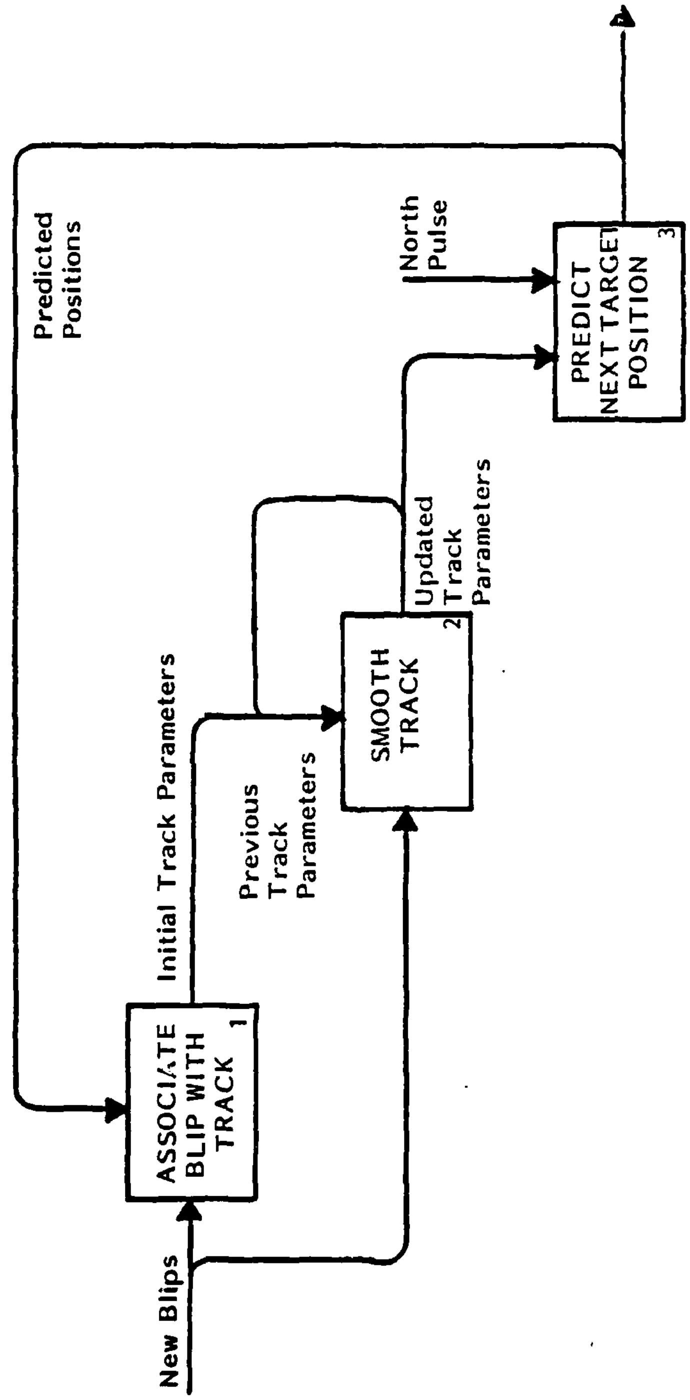


Figure 2. Track Targets

Discussion

The selection of a tasking structure for an embedded computer program is one of the most basic design decisions that must be made. The Ada tasking features offer a variety of possible tasking structures for any given problem. There is currently no satisfactory general method of selecting a good tasking structure from the range of possibilities.

The possible task structures for the target tracking system include:

a. a single task handling all the work

b. division of the scan into sectors and assigning a separate task to handle all functions within each sector

c. assigning a separate task to each major function (associate, smooth, and predict)

d. assigning a separate task to each new blip that enters the system

e. assigning a separate task to each active track in the system.

The structures a., b., and c. are multi-threaded structures in which the actions for particular blips and tracks are explicitly interleaved by the program. The structures in d. and e. are single-threaded for actions dealing with particular blips and tracks, respectively.

The main issues to be considered in selecting a tasking structure are:

- safety: the degree to which it properly implements the required functions without deadlock or other unacceptable effects
- clarity: the degree to which the program can be understood
- efficiency: the throughput or responsiveness that can be expected

In general, multi-threaded approaches tend to be more efficient while sacrificing clarity, whereas single-threaded approaches exhibit the opposite characteristic. Depending on the particular application the loss of efficiency of a single-threaded approach may be trivial or serious.

The particular approach explored in this case study is a combination of approach c. and e. above. There is a single, multi-threaded association task coupled with separate tasks for each track to carry out the smoothing and prediction functions.

2. SOLUTION

Solution Outline

The solution presented in this case study starts with a main program which declares generally used data types and structures as well as the actual processing tasks. The main program itself carries out no actions; all the work is done in the declared tasks. Two of the three referenced tasks (or task types) are shown in the case study -- ASSOCIATE and TRACK_PROCESS. The ASSOCIATE task correlates new blips with existing tracks or creates new tracks when no correlation is found. There is one task of type TRACK_PROCESS for each track, which carries out the smoothing and prediction function for that track. The actual algorithms for smoothing and prediction are not shown in this case study. The RADAR_INTERFACE task is not shown since it deals with the low-level details of interfacing with the radar hardware. It calls the ASSOCIATE task with each blip received from the radar and calls each TRACK_PROCESS task when the north pulse is received from the radar.

The primary data structure is an array of records representing tracks. Each record contains the data about a particular track. The record includes as a component the TRACK_PROCESS task for the track.

Detailed Solution

```
procedure MAIN is
     subtype NON_NEGATIVE_INTEGER is INTEGER range 0..INTEGER'LAST;
     type REAL is digits 8;
     type BLIP_TYPE is
          record
               RANGE      : REAL;
               BEARING    : REAL;
               RANGE_RATE : REAL;
          end record;

     type TRACK_ID_TYPE is range 1..200;

     task type TRACK_PROCESS is
          entry ACTIVATE ( TRACK_ID : TRACK_ID_TYPE );
          entry NEW      ( BLIP     : BLIP_TYPE );
          entry NORTH_PULSE;
     end TRACK_PROCESS;

     type TRACK_PARAM_TYPE is
          record
               X, Y, PHI, THETA, SMOOTH_INDEX : REAL;
          end record;

     type TRACK_TYPE is
          record
               ACTIVE  : BOOLEAN := FALSE;
               READY   : BOOLEAN;
               PROCESS : TRACK_PROCESS;
               PARMS   : TRACK_PARAM_TYPE;
          end record;

     task ASSOCIATE is
          entry RADAR ( BLIP : BLIP_TYPE );
          entry READY ( ID   : TASK_ID_TYPE );
     end ASSOCIATE;

     task RADAR_INTERFACE;  -- calls ASSOCIATE with new blips and each
                            -- active TRACK_PROCESS with NORTH_PULSE

     TRACK : array ( TRACK_ID_TYPE ) of TRACK_TYPE;

     task body ASSOCIATE       is separate;
     task body TRACK_PROCESS   is separate;
     task body RADAR_INTERFACE is separate;

begin
     null;  -- all work is done in the dependent tasks
end MAIN;
```

```
separate ( MAIN )

task body TRACK_PROCESS is
     MY_ID    : TRACK_ID_TYPE;
     NEW_BLIP : BLIP_TYPE;
     procedure SMOOTH ( BLIP     : BLIP_TYPE;
                        TRACK_ID : TRACK_ID_TYPE ) is separate;
     procedure PREDICT_NEW_POSITION ( TRACK_ID : TRACK_ID_TYPE )
                                              is separate;
     MISSED_BLIPS   : NON_NEGATIVE_INTEGER;
     MISS_THRESHOLD : constant NON_NEGATIVE_INTEGER := 5;

begin     -- TRACK_PROCESS
     loop -- forever
          accept ACTIVATE ( TRACK_ID : TRACK_ID_TYPE ) do
               MY_ID := TRACK_ID;
          end ACTIVATE;

          loop
               ASSOCIATE.READY ( MY_ID );
               select
                    accept NEW ( BLIP : BLIP_TYPE ) do -- have blip
                                                       -- this scan
                         NEW_BLIP := BLIP;
                    end NEW;
                    SMOOTH ( NEW_BLIP, MY_ID );
                    PREDICT_NEW_POSITION ( MY_ID );
                    MISSED_BLIPS := 0;
                    accept NORTH_PULSE;
               or
                    accept NORTH_PULSE;                -- no blip
                                                       -- this scan
                    PREDICT_NEW_POSITION ( MY_ID );
                    MISSED_BLIPS := MISSED_BLIPS + 1;
               end select;
          exit when MISSED_BLIPS > MISS_THRESHOLD;
          end loop;

          TRACK ( MY_ID ).ACTIVE := FALSE;
     end loop;  -- forever
end TRACK_PROCESS;
```

```
separate ( MAIN )
task body ASSOCIATE is
     NEW_BLIP          : BLIP_TYPE;
     BEST_MATCH,MATCH  : NON_NEGATIVE_INTEGER;
     MATCH_THRESHOLD   : constant NON_NEGATIVE_INTEGER := 100;
     BEST_TRACK_ID     : TRACK_ID_TYPE;
     function DEGREE_OF_FIT ( BLIP         : BLIP_TYPE;
                              TRACK_ID     : TRACK_ID_TYPE )
              returns NON_NEGATIVE_INTEGER is separate;
     procedure INITIATE_NEW_TRACK ( BLIP : BLIP_TYPE ) is separate;

begin -- ASSOCIATE
     loop -- forever
          select
               accept RADAR ( BLIP : BLIP_TYPE ) do
                    NEW_BLIP := BLIP;
               end RADAR;

               BEST_MATCH := 0;
               for TRACK_ID in TRACK_ID_TYPE
               loop
                    if TRACK ( TRACK_ID ).ACTIVE and
                       TRACK ( TRACK_ID ).READY then
                         -- try match with TRACK_ID
                         MATCH := DEGREE_OF_FIT ( NEW_BLIP, TRACK_ID );
                         if MATCH > MATCH_THRESHOLD and
                            MATCH > BEST_MATCH then
                              -- found best match so far
                              BEST_MATCH := MATCH;
                              BEST_TRACK_ID := TRACK_ID;
                         end if;
                    end if;
               end loop;

               if BEST_MATCH > MATCH_THRESHOLD then
                    -- found good match
                    TRACK ( BEST_TRACK_ID ).READY := FALSE;
                    TRACK ( BEST_TRACK_ID ).PROCESS.NEW ( NEW_BLIP );
               else  --  no match
                    INITIATE_NEW_TRACK ( NEW_BLIP );
               end if;
          or
               accept READY ( TRACK_ID ) do
                    TRACK ( TRACK_ID ).READY := TRUE;
               end READY;
          end select;
     end loop;

end ASSOCIATE;
```

3. EPILOGUE

Because of the central importance of the selection of tasking structure to an Ada software design methodology, a good deal of additional work is warranted. In particular, more general conclusions about the relative advantages of single-threaded versus multi-threaded designs in specific cases would be desirable. The aim would be to find general conditions under which a particular approach is acceptable. In addition, further study could provide guidelines for selecting the aspects of a problem around which to structure the tasks. In the example in this case study, the question was whether the tasks should be organized around functions, sectors, blips, or tracks.

As a first step in this study, additional solutions to this particular case study problem could be developed. Developing additional solutions could be an enlightening student problem.

THIS PAGE INTENTIONALLY LEFT BLANK

UART: EXPRESSING HARDWARE DESIGN IN ADA

1. BACKGROUND

Case Study Objective

To express hardware functionality in Ada. To express timing constraints in Ada.

Designer's Problem

The output of the system design phase should be an expression of the system's design prior to any decisions about hardware and software. Inevitably, the designer recognizes certain pieces in the design as likely hardware candidates, yet these too must be expressed in the system design language, thereby avoiding premature hardware/software tradeoff evaluations.

When the system design language is Ada, which constructs can best be used in a hardware specification? How does one model timing requirements, real-time events, parallel operations? One of the contractors had attempted to represent the functionality of an universal asynchronous receiver transmitter (UART) in Ada.

Discussion

What is important to recognize here is that there is a two-sided interface. On the one hand, the UART is a chip that fits on a board, a black box that transforms serial or parallel input into parallel or serial output respectively. On the other hand, from the UART's point of view, it expects certain inputs and connections to ensure its correct functioning. The hardware designers are aware of both facets in order to install such devices properly. Software designers are in an analogous situation, as explained below (see Figure 1). The physical UART must be first installed and some of the pins must be connected to constant voltages that provide operating options. This "inside looking out" view can be captured by making the UART a generic package. The generic parameters constitute in a way a model of the socket into which the UART is plugged.

2. DETAILED EXAMPLE

Example Problem Statement

A hardware device can be described from two points of view -- behavioral (black box) or structural (glass box). In the behavioral or black box approach the device is described strictly in terms of its input-output behavior including the functional and timing relationships that exist among the inputs and outputs. There is no intended

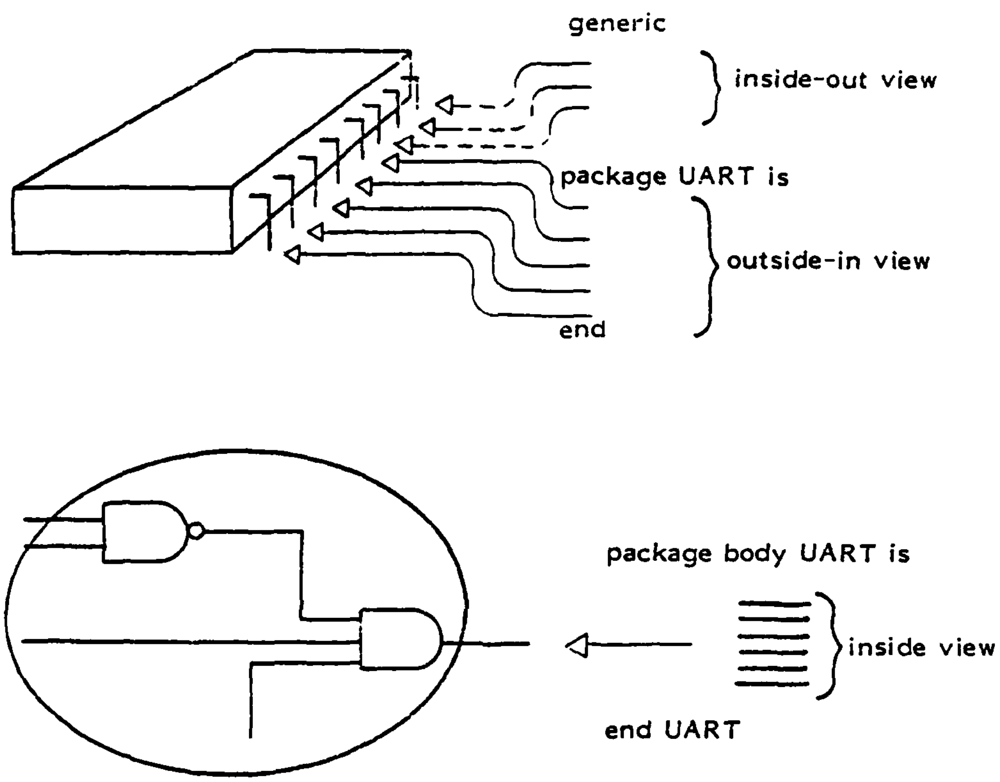


Figure 1. Analogy Between Hardware and Software Views

correlation between the parts of a behavioral model of a device and the physical components of the device. On the other hand, in the structural or glass box approach the device is described in terms of its actual components and their interconnections.

The essence of top-down hardware design is an alternation between these two view points. First a behavioral description of a device as a whole is developed. Then the designer synthesizes and describes a structure -- on interconnection of subcomponents -- for the device. The process is then repeated by developing behavioral specifications at the subcomponent level. The integrity of the design is verified by demonstrating that the specified interconnection of subcomponents, each with a specified behavior, is equivalent (in terms of input-output relationship) to the behavioral specifications for the device as a whole.

In this case study the entire UART is described as a structural interconnection of four components, as shown in Figure 2. Each of these four components is further described from the behavioral viewpoint.

In the Ada specification of the UART, each component is represented as a task. The entry calls together with their parameters define the interconnections among the components. The generic parameters to the UART represent the control parameters connection on the diagram (see Figure 1). The externally available calls on RECEIVE_CHAR, TRANSMIT CHAR, and EXTERNAL_RESET represent the control commands.

The UART performs two services:

1. It accepts serial input, assembles the incoming bits into a character and makes the character available on its parallel output port.

2. It accepts a character on its parallel input port and transmits it bit by bit on a serial output line.

The UART can be set for even, odd or no parity, for 5, 6, 7, or 8 bit characters, and for 1 or 2 stop bits. Additionally, the receive and transmit operations work in parallel, and their corresponding baud rates are set individually (i.e., the baud rates may differ, though the bit format options may not).

The following protocol is used to delimit a character. When there is no data on the line, the voltage level is kept at a logical 1, in the mark condition. A single start bit is recognized when the line drops to a logical 0, the space condition. The next n data bits are the character, where n is the number of bits per character. If the UART is set for even or odd parity, the next bit on the line is the parity bit. The last bit on the line is the stop bit (two stop bits, if the UART is set with that option), which is always the line being in the mark condition. The line now remains in the mark condition until another character is put on the line.

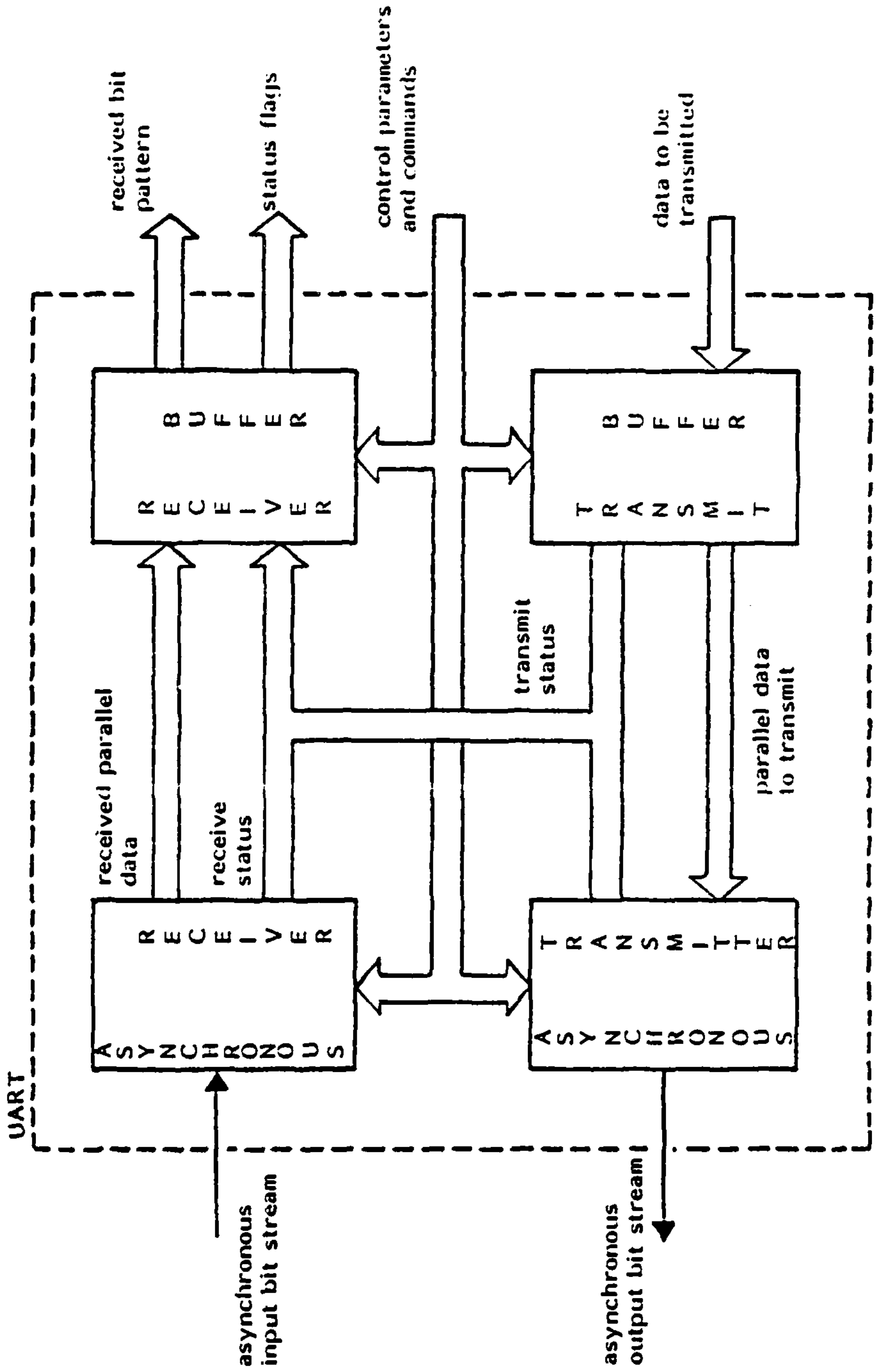


Figure 2. Top Level Structural View of the UART

The receiver portion of the UART detects 3 errors in addition to setting a flag to indicate the presence of a character on the output port. The errors are: data overrun, parity error, and framing error. Data overrun occurs when the previous character placed in the parallel output port is not read before the current character is placed there. Parity error occurs when the parity read from the serial input disagrees with the parity that the UART computes for the character it received. Framing error means that a stop bit failed to come across the line.

The transmitter portion of the UART sets a flag when it is ready to accept another character on its parallel input lines. It sets another flag when it has finished transmitting a character.

Both the receiver and transmitter portion of the UART have internal clocks whose frequency is sixteen times the set data rate, where the data or bit rate is defined as the reciprocal of the baud rate. Thus a bit is transmitted (the line is held in the mark or space condition) for a duration of sixteen clock times or one bit time (the data rate). From the receiver viewpoint, the logical value of a bit is determined from sampling the serial input line in the middle of the time interval during which a bit is sent. (See Figure 3.)

The UART has an external reset line which resets the parallel input lines and all flags in both receive and transmit sections.

Solution Outline

This example presents the receiver section of the UART in depth while it sketches the transmitter specification.

The UART is a package which, from the outside world's point of view:

1. accepts serial input (asynchronous input bit stream);
2. delivers a character and four status flags;
3. accepts parallel input (a character);
4. delivers a bit stream (asynchronous output bit steam).

The package body implements the processing of bits into a character, the setting of appropriate status flags, the placing of the character on parallel output lines, the bitwise transmission of a character read from parallel input lines, and the setting of appropriate transmission status flags. Figure 4 shows a structure chart for the receiver section of a UART.

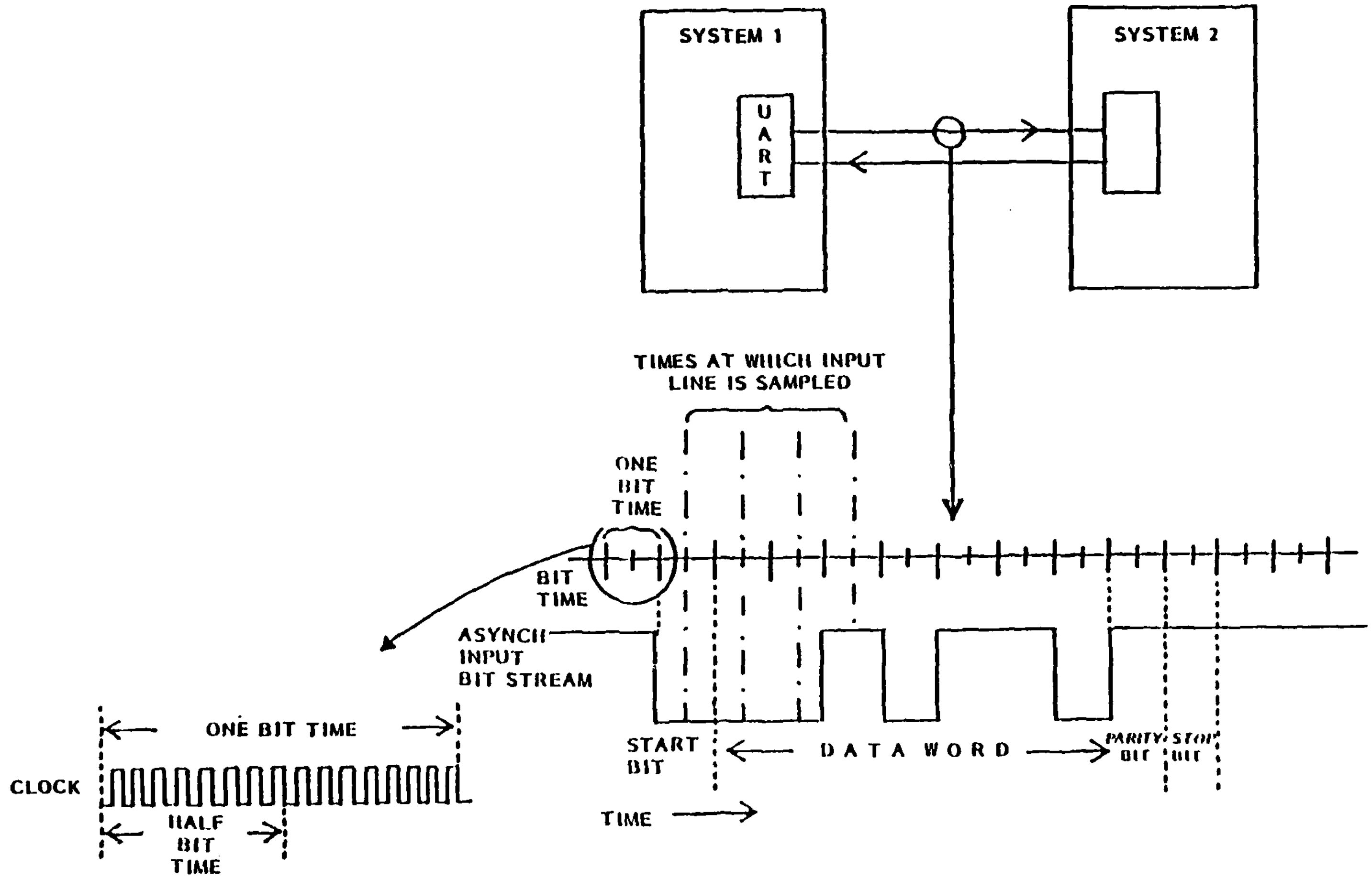


Figure 3. Timing Diagram for UART

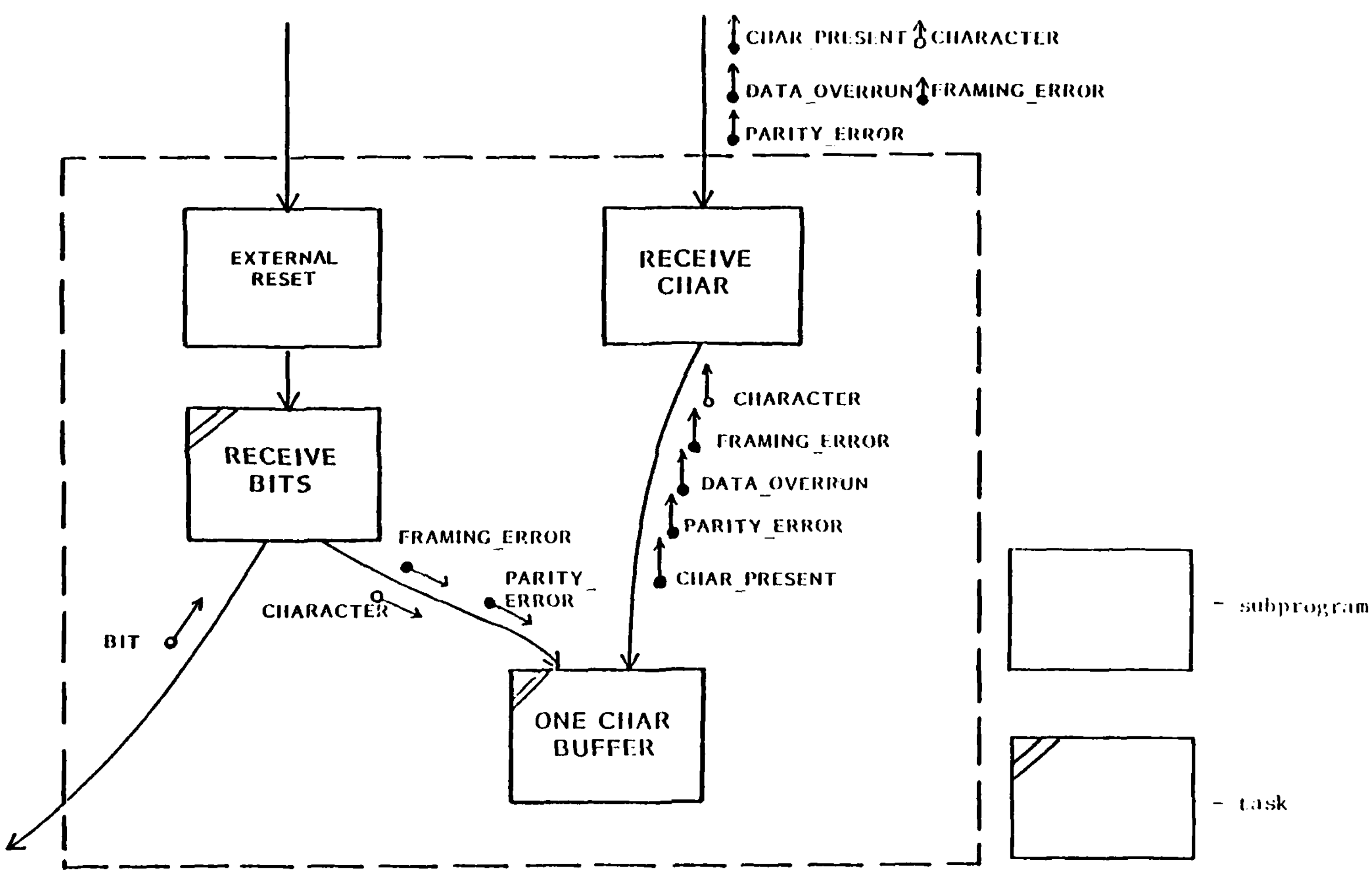


Figure 4. Structure Chart for UART

SOFTECH

Generic parameters are used to specify the bit format options, the receive and transmit baud rates, and the asynchronous serial input and output lines. Generic parameters serve several purposes. They capture design options, such as the bit format and the baud rates that may vary from one UART to another. They also postpone design decisions, such as deciding what voltages will represent the space and mark conditions and how the serial input and output lines will be represented.

Within the receive section of a UART, there are two events: bits from the serial input line are read and assembled into a character, and then the character is put in a buffer from which the character and status flags are read out. Tasks are used to model these two events. In order to model the actual decoupling of the assembling of the character from its being read out, it is necessary to have the buffering task. Because the bit assembling process is concurrent and ongoing, it is also modeled as a task.

A UART operates in real time, and this example presents a method to express timing constraints. An underlying assumption is made in this specification, namely that all statements execute in zero (0) time, with the exception of the delay statement, which "advances" time by executing a delay of the specified duration. This concept is critical to the specification because it enables the designer to state what events occur at which instants in time. Named constants for time intervals are introduced in order to enhance readability.

Detailed Solution

```
package UART_DEFS is

     type PTY is (EVEN,ODD);
     PARITY_STATUS_FLAG           : BOOLEAN := FALSE;
     FRAMING_ERROR_FLAG           : BOOLEAN := FALSE;

     CLOCK_TIMES_PER_BIT_TIME     : constant := 16;
     CLOCK_TIMES_PER_HALF_BIT_TIME: constant := CLOCK_TIMES_PER_BIT_TIME / 2;

     procedure RESET_PARITY_STATUS_FLAG;

     procedure SET_PARITY_STATUS_FLAG;

     procedure FLIP(PARITY: in out PTY);

     procedure RESET_FRAMING_ERROR_FLAG;

     procedure SET_FRAMING_ERROR_FLAG;

end UART_DEFS;
```

```
package body UART_DEFS is

     procedure RESET_PARITY_STATUS_FLAG is
     begin
          PARITY_STATUS_FLAG := FALSE;
     end RESET_PARITY_STATUS_FLAG;

     procedure SET_PARITY_STATUS_FLAG is
     begin
          PARITY_STATUS_FLAG := TRUE;
     end SET_PARITY_STATUS_FLAG;

     procedure FLIP(PARITY: in out PTY) is
     begin
          if PARITY = EVEN then
               PARITY := ODD;
          else
               PARITY := EVEN;
          end if;
     end FLIP;

     procedure RESET_FRAMING_ERROR_FLAG is
     begin
          FRAMING_ERROR_FLAG := FALSE;
     end RESET_FRAMING_ERROR_FLAG;

     procedure SET_FRAMING_ERROR_FLAG is
     begin
          FRAMING_ERROR_FLAG := TRUE;
     end SET_FRAMING_ERROR_FLAG;

end UART_DEFS;
```

```
generic

    -- LEVEL is the number of bits in the character, indicating the
    -- type of code, such as Ascii, Baudot, etc.
    -- RECEIVE_BAUD_RATE indicates the incoming baud rate
    -- TRANSMIT_BAUD_RATE indicates the outgoing baud rate
    -- WITH_PARITY indicates whether parity will be transmitted or
    -- received.  Either parity is present both for receive and for
    -- transmit or it is absent.
    -- EVEN_PARITY indicates whether the parity will be even or odd.
    -- The type BIT models the voltage levels used for logical 0 and
    -- logical 1.
    -- ASYNCH_INPUT_BIT_STREAM models the hardwired line from which the
    -- incoming bits will be read.
    -- ASYNCH_OUTPUT_BIT_STREAM models the hardwired line on which the
    -- outgoing bits will be sent.
    -- SPACE represents a logical 0.
    -- MARK represents a logical 1.  SPACE and MARK are the names
    -- traditionally used in asynchronous communication to represent
    -- these logical values.

    LEVEL              : NATURAL range 5..8;
    RECEIVE_BAUD_RATE  : in DURATION;
    TRANSMIT_BAUD_RATE : in DURATION;
    WITH_PARITY        : in BOOLEAN;
    EVEN_PARITY        : in BOOLEAN;
    NUMBER_OF_STOP_BITS: NATURAL range 1..2;

    type BIT is (<>);
    with function ASYNCH_INPUT_BIT_STREAM return BIT;
    with procedure ASYNCH_OUTPUT_BIT_STREAM(XMIT_BIT: in BIT);

    SPACE              : in BIT;
    MARK               : in BIT;
```

```
package UART is

     type BIT_SEQUENCE is array (0..7) of BIT;
       --this array represents the 8 parallel output lines onto which
       -- the UART transfers the character it reads from the incoming
       -- bit stream.  Note that the least significant bit is
       -- transmitted first, and it is shifted through the lines (i.e.
       -- array positions) from the last position to the first position.
       -- thus BIT_SEQ(0) contains the least significant bit and
       -- BIT_SEQ(LEVEL-1) contains the most significant bit, where
       -- BIT_SEQ is an object declared to be of type BIT_SEQUENCE.

     type R_CHARACTER_CONDITIONS is (DATA_OVERRUN,
                                     CHAR_PRESENT,
                                     PARITY_ERROR,
                                     FRAMING_ERROR);

     type R_CONDITION_SET is array (R_CHARACTER_CONDITIONS) of BOOLEAN;
       --R_CONDITION_SET holds the status flags which are associated with
       -- the data being transmitted or received.

     type T_CHARACTER_CONDITIONS is (TRANSMIT_FINISHED,
                                     TRANSMIT_BUFFER_EMPTY);

     type T_CONDITION_SET is array (T_CHARACTER_CONDITIONS) of BOOLEAN;

     procedure EXTERNAL_RESET;

     procedure RECEIVE_CHAR(DATA  : out BIT_SEQUENCE;
                            STATUS: out R_CONDITION_SET);
       --allows the user to read out in parallel the character which was
       -- input serially to the UART.

     procedure TRANSMIT_CHAR(DATA  : in BIT_SEQUENCE;
                             STATUS: out T_CONDITION_SET);
       --allows the user to give the UART the character which should be
       -- then transmitted out serially.

end UART;
```

```
with UART_DEFS; use UART_DEFS;
package body UART is

     --In the time domain, a bit is defined as the reciprocal of the
     -- baud rate.  A clock time is defined as a fraction, usually one
     -- sixteenth, of one bit time.  Thus, by sampling the line at some
     -- instant in a time interval lasting sixteen clock times, a
     -- voltage is obtained, and this voltage is what determines the
     -- logical value of the bit.  Because the UART can receive bits at a
     -- different rate than it transmits them, there are bit time definitions
     -- both for the receive and for the transmit baud rates.  Rather than
     -- use the literal constants 16 and 8 (representing the number of
     -- clock times per bit time or half bit time respectively), constants
     -- have been defined in a utility package (UART_DEFS), thereby
     -- facilitating the maintenance of this Ada model of a UART.

     ONE_BIT_R_TIME            : constant DURATION :=
               DURATION(1.0 / RECEIVE_BAUD_RATE);
     HALF_BIT_R_TIME           : constant DURATION :=
               DURATION(ONE_BIT_R_TIME / 2.0);
     ONE_AND_A_HALF_BIT_R_TIME: constant DURATION :=
               DURATION(ONE_BIT_R_TIME * 1.5);
     R_CLOCK_TIME              : constant DURATION :=
               DURATION(ONE_BIT_R_TIME / DURATION(CLOCK_TIMES_PER_BIT_TIME));

     ONE_BIT_T_TIME            : constant DURATION :=
               DURATION(1.0 / TRANSMIT_BAUD_RATE);
     HALF_BIT_T_TIME           : constant DURATION :=
               DURATION(ONE_BIT_T_TIME / 2.0);
     ONE_AND_A_HALF_BIT_T_TIME: constant DURATION :=
               DURATION(ONE_BIT_T_TIME * 1.5);
     T_CLOCK_TIME              : constant DURATION :=
               DURATION(ONE_BIT_T_TIME / DURATION(CLOCK_TIMES_PER_BIT_TIME));

     BIT_SEQ: BIT_SEQUENCE;

     task ASYNCHRONOUS_RECEIVER is
          entry EXTERNAL_RESET;      --External reset resets all the
     end ASYNCHRONOUS_RECEIVER;      -- condition flags to FALSE and
                                     -- zeroes out the bit sequence on the
                                     -- parallel input lines.  It then
                                     -- allows the UART to proceed
                                     -- normally, i.e. to await the user's
                                     -- request to receive or transmit
                                     -- data.  The implementation is left
                                     -- as an exercise to the reader.

     task ASYNCHRONOUS_TRANSMITTER is
          entry EXTERNAL_RESET;
     end ASYNCHRONOUS_TRANSMITTER;

     task RECEIVER_BUFFER is
          entry RECEIVE_BITS(BIT_PATTERN  : in BIT_SEQUENCE;
                             PARITY_ERROR : in BOOLEAN;
                             FRAMING_ERROR: in BOOLEAN);
          entry FETCH_CHAR(BIT_PATTERN : out BIT_SEQUENCE;
                           STATUS_FLAGS: out R_CONDITION_SET);
     end RECEIVER_BUFFER;

     task TRANSMITTER_BUFFER is
          entry SEND_CHAR(DATA  : in BIT_SEQUENCE;
                          STATUS: out T_CONDITION_SET);
     end TRANSMITTER_BUFFER;
```

```
function BIT_TO_PTY(SIGNAL: BIT) return PTY is
begin
     if SIGNAL = SPACE then
          return EVEN;
     else
          return ODD;
     end if;
end BIT_TO_PTY;


function PTY_TO_BIT(PRTY: PTY) return BIT is
begin
     if PRTY = EVEN then
          return SPACE;
     else
          return MARK;
     end if;
end PTY_TO_BIT;


procedure EXTERNAL_RESET is

--This procedure allows the user to issue an external reset command to
-- the UART and reset all the condition flags and the parallel lines
-- containing the character to be transmitted.  The UART then goes
-- back to its normal operating loop of looking for characters to
-- receive or transmit.

begin
     ASYNCHRONOUS_RECEIVER.EXTERNAL_RESET;
     ASYNCHRONOUS_TRANSMITTER.EXTERNAL_RESET;
end EXTERNAL_RESET;
```

```
task body ASYNCHRONOUS_RECEIVER is

     PARITY: PTY;

begin

CONSTRUCT_AND_SEND_CHAR:
     loop                                   --infinite loop

     --By convention, a logical 1 or mark is on the asynchronous
     -- input line until the beginning of transmitted data.  The
     -- line is checked at every clock cycle, until a start signal
     -- is detected, when this line transits to the space condition
     -- or logical 0 for a duration of one bit time.
     --For timing purposes, assume that instructions execute in
     -- zero time.  For implementation, delays must be adjusted for
     -- instruction execution times.
     --Because of the possibility of a false start (the line
     -- temporarily transits to the space condition) or of noise
     -- received during the transmission of the start bit (the line
     -- temporarily transits to the mark condition), the UART
     -- checks that the line is in the space condition for a half
     -- bit time, allowing for one noise spike.  In other words,
     -- the line must be in the space condition for at least one
     -- clock time less than the number of clock times in a half
     -- bit time.

     declare
          type PULSE_SEQUENCE is
                  array (1..CLOCK_TIMES_PER_HALF_BIT_TIME) of BIT;
          subtype INTEGER_RANGE is range 0 .. PULSE_SEQUENCE'LAST;
          INPUT_IMAGE: PULSE_SEQUENCE :=
                  (INPUT_IMAGE'FIRST..INPUT_IMAGE'LAST => MARK);
          COUNT      : INTEGER_RANGE := 0;
          I          : INTEGER_RANGE := INPUT_IMAGE'FIRST;
     begin
     WAIT_FOR_START_SIGNAL:
          loop
               COUNT := 0;
               INPUT_IMAGE(I) := ASYNCH_INPUT_BIT_STREAM;
               for J in INPUT_IMAGE'RANGE
               loop
                    if INPUT_IMAGE(J) = SPACE then
                         COUNT := COUNT + 1;
                    end if;
               end loop;
          exit when COUNT >= INPUT_IMAGE'LAST - 1;
               I := (I mod CLOCK_TIMES_PER_HALF_BIT_TIME) + 1;
               delay R_CLOCK_TIME;
          end loop WAIT_FOR_START_SIGNAL;
     end;
```

```
     --A transition has been detected, i.e. the input line has
     -- dropped to 0.  Initialize the parity to EVEN (assume
     -- as a default that an all 0 word has even parity).
     -- Reset the lines onto which the UART will transfer the
     -- incoming serial data, i.e. zero the bit sequence array.
     -- Reset the parity error warning flag as well as the
     -- framing error flag.  One bit time following receipt of
     -- the start bit, the first data bit will come over the
     -- line.  To minimize the effect of clock drift between
     -- the sender's and the asynchronous receiver's clocks,
     -- the data bits will be read in the "middle" of the
     -- incoming bit.

     BIT_SEQ := (0..BIT_SEQ'LAST => SPACE);
     PARITY := EVEN;
     RESET_PARITY_STATUS_FLAG;
     RESET_FRAMING_ERROR_FLAG;
     delay ONE_AND_A_HALF_BIT_R_TIME;

     --Read in n bits, corresponding to some predetermined
     -- n-level code. If the EVEN or ODD parity option is in
     -- effect, calculate the parity and set a warning flag if
     -- this parity differs from the transmitted one.

ACCEPT_BIT_STREAM:
     for I in 0..LEVEL-1
     loop
     --Shift bits down one line and accept most recent bit
     -- from serial input line.
          BIT_SEQ(0..BIT_SEQ'LAST-1) := BIT_SEQ(1..BIT_SEQ'LAST);
          BIT_SEQ(BIT_SEQ'LAST) := ASYNCH_INPUT_BIT_STREAM;
          if BIT_SEQ(BIT_SEQ'LAST) = MARK then
               FLIP(PARITY);
          end if;
          delay ONE_BIT_R_TIME;
     end loop ACCEPT_BIT_STREAM;
SHIFT_LSB_TO_LSB_POSITION:
     for I in LEVEL..BIT_SEQ'LAST
     loop
          BIT_SEQ(0..BIT_SEQ'LAST-1) := BIT_SEQ(1..BIT_SEQ'LAST);
          BIT_SEQ(BIT_SEQ'LAST) := SPACE;
     end loop SHIFT_LSB_TO_LSB_POSITION;
     if WITH_PARITY then
          if EVEN_PARITY then
               if PARITY /= BIT_TO_PTY(ASYNCH_INPUT_BIT_STREAM) then
                    SET_PARITY_STATUS_FLAG;
               end if;
          else  --not even implies odd
               if PARITY = BIT_TO_PTY(ASYNCH_INPUT_BIT_STREAM) then
                    SET_PARITY_STATUS_FLAG;
               end if;
          end if;
          delay ONE_BIT_R_TIME;
     end if;
```

```
          --Now that a bit sequence has been read in, one must check
          -- that the serial input line returns to the mark condition
          -- during the next bit time.  Should this fail to happen
          -- the framing error flag is set.  Should the line return
          -- to the mark condition, then nothing happens.  Thus, if
          -- there already existed a framing error, the error flag
          -- remains set or, in worst case, gets set again.  A
          -- framing error would be handled externally to the asyn-
          -- chronous receiver task.

     FRAMING_BIT_DETECTION:
          declare
               NO_FRAMING_ERROR: exception;
               FRAMING_ERROR: exception;
          begin
               for I in 1..CLOCK_TIMES_PER_BIT_TIME
               loop
                    if ASYNCH_INPUT_BIT_STREAM = MARK then
                         raise NO_FRAMING_ERROR;
                    end if;
                    delay R_CLOCK_TIME;
               end loop;
               raise FRAMING_ERROR;
          exception
               when FRAMING_ERROR =>
                    SET_FRAMING_ERROR_FLAG;
               when NO_FRAMING_ERROR =>
                    null;
          end FRAMING_BIT_DETECTION;

          --The character is now ready for whomever wants to fetch
          -- it.  It is put into a data holding buffer, from
          -- where any one can retrieve it.

          RECEIVER_BUFFER.RECEIVE_BITS(BIT_PATTERN => BIT_SEQ,
                                       PARITY_ERROR => PARITY_STATUS_FLAG,
                                       FRAMING_ERROR =>FRAMING_ERROR_FLAG);

     end loop CONSTRUCT_AND_SEND_CHAR;

end ASYNCHRONOUS_RECEIVER;
```

```
task body ASYNCHRONOUS_TRANSMITTER is

     PARITY: PTY;
     DATA: BIT_SEQUENCE;
     STATUS: T_CONDITION_SET;

     procedure SEND_BIT(THE_BIT: in BIT) is
     begin
          ASYNCH_OUTPUT_BIT_STREAM(THE_BIT);
          delay ONE_BIT_T_TIME;
     end SEND_BIT;

begin

     loop
           TRANSMITTER_BUFFER.SEND_CHAR(DATA,STATUS);
           STATUS(TRANSMIT_FINISHED) := FALSE;
           SEND_BIT(SPACE); --send start signal
           PARITY := EVEN;
           for I in 0..LEVEL-1
           loop
               SEND_BIT(DATA(I));
               if DATA(I) = MARK then
                    FLIP(PARITY);
               end if;
          end loop;
          if WITH_PARITY then
               if EVEN_PARITY then
                    SEND_BIT(PTY_TO_BIT(PARITY));
               else
                    FLIP(PARITY);
                    SEND_BIT(PTY_TO_BIT(PARITY));
               end if;
          end if;
          for I in 1..NUMBER_OF_STOP_BITS
          loop
               SEND_BIT(MARK);
          end loop;
          STATUS(TRANSMIT_FINISHED) := TRUE;
     end loop;
end ASYNCHRONOUS_TRANSMITTER;
```

```
task body RECEIVER_BUFFER is

    STATUS: R_CONDITION_SET;
    DATA: BIT_SEQUENCE;

begin

    loop
        select
            accept RECEIVE_BITS(BIT_PATTERN  : in BIT_SEQUENCE;
                                PARITY_ERROR : in BOOLEAN;
                                FRAMING_ERROR: in BOOLEAN) do
                if STATUS(CHAR_PRESENT) then
                    STATUS(DATA_OVERRUN) := TRUE;
                end if;
                STATUS(PARITY_ERROR) := PARITY_ERROR;
                STATUS(FRAMING_ERROR) := FRAMING_ERROR;
                STATUS(CHAR_PRESENT) := TRUE;
                DATA := BIT_PATTERN;
            end RECEIVE_BITS;
        or
            accept FETCH_CHAR(BIT_PATTERN : out BIT_SEQUENCE;
                              STATUS_FLAGS: out R_CONDITION_SET) do
                STATUS(DATA_OVERRUN) := FALSE;
                STATUS(CHAR_PRESENT) := FALSE;
                BIT_PATTERN := DATA;
            end FETCH_CHAR;
        end select;
    end loop;

end RECEIVER_BUFFER;
```

```
     task body TRANSMITTER_BUFFER is separate;


     procedure RECEIVE_CHAR(DATA: out BIT_SEQUENCE;
                            STATUS: out R_CONDITION_SET) is

     begin
          RECEIVER_BUFFER.FETCH_CHAR(DATA,STATUS);
          while not STATUS(CHAR_PRESENT)
          loop
               RECEIVER_BUFFER.FETCH_CHAR(DATA,STATUS);
          end loop;
     end SEND_CHAR;



     procedure TRANSMIT_CHAR(CHAR  : in CHARACTER;
                             STATUS: out T_CONDITION_SET) is separate;

end UART;
```

3. EPILOGUE

The UART specification presented in the detailed solution is for an uninstalled device. The physical installation of the device, i.e, the soldering of the pin connections, is equivalent to instantiating the generic package UART with the desired bit format option, baud rates, voltages, etc.

```
package MY_UART_SPECS is
     function PIN_20_SERIAL_INPUT return BOOLEAN;
     procedure PIN_25_SERIAL_OUTPUT(XMIT: BOOLEAN);
     type BIT_VOLTAGE is (PLUS_12_V, MINUS_12_V);
end MY_UART_SPECS;
with UART, MY_UART_SPECS; use MY_UART_SPECS;
package MYUART is new UART(LEVEL                     => 7,
                           RECEIVE_BAUD_RATE         => 300.0,
                           TRANSMIT_BAUD_RATE        => 110.0,
                           WITH_PARITY               => TRUE,
                           EVEN_PARITY               => TRUE,
                           NUMBER_OF_STOP_BITS       => 1,
                           BIT                       => BIT_VOLTAGE,
                           ASYNCH_INPUT_BIT_STREAM   => PIN_20_SERIAL_INPUT,
                           ASYNCH_OUTPUT_BIT_STREAM  => PIN_25_SERIAL_OUTPUT
                           SPACE                     => PLUS_12_V,
                           MARK                      => MINUS_12_V);
```

This example was written from the hardware design point of view. A different viewpoint may be taken, where the designer also has the ability to specify interfaces without knowing where the hardware/software boundary lies. Using Ada effectively reinforces the principle of completing the system design prior to partitioning the hardware and software of the system. Consider a computer system which contains an asynchronous communication subsystem, consisting of some boards and a device driver. This entire subsystem can be specified in Ada, regardless of what part of the driver is hardware and what portion software. Thus the allocation of hardware functions is deferred until a later time. For instance, one could have constructed an abstract model of a UART as viewed by the software at the device driver interface.

Delay statements model time in the asynchronous transmitting and receiving of data. An alternative implementation could use accept statements, where an interval of time is marked by accepting the number of clock pulses in that period. This code shows how a delay in fact works: the hardware waits for a transition on its clock input line:

```
for I in 1 .. 16           -- Equivalent to waiting for a duration
loop                            -- of 16 clock times,
     accept CLOCK_PULSE;   -- or one bit time, i.e., delay
end loop;                       -- ONE_BIT_R_TIME
```

The idea of using Ada as a hardware specification language is interesting on several counts. It serves as a standardized communication tool between the system designer and the hardware designer. Another benefit is that standard interface specifications can be written without regard for the internal hardware/software partition. Furthermore, this capability means that these devices and interfaces may be simulated, thereby enhancing the design process.

The Ada functional specification of a UART illustrates both Ada features and Ada hardware design. Based on this example, there are several topics which can be pursued as exercises or classroom discussion:

- Complete the implementation of the transmit portion of the UART.

- Assume that an external reset can be received at any time, and that it must be processed as soon as possible. Implement the external reset operation. (Hint: an external reset can be accepted between any two incoming or outgoing bits. Note that after an external reset is processed, both the receiver and the transmitter should be waiting to receive the next start bit or the next character respectively.)

- Design a Universal Synchronous Receiver Transmitter in Ada. (Hint: the timing requirements are best modeled by the "accept CLOCK_PULSE" construct.)

- Specify an abstract UART from the device driver software's point of view.

Section 4

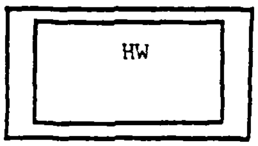
DETAILED DESIGN PHASE

4.1 Purpose

The detailed design phase develops the algorithms for the software identified in the previous design phase. A combination of Ada as a program design language (PDL) and graphical tools is used at this level.

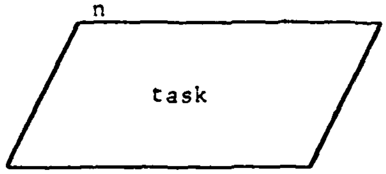
4.2 Graphical Tools

The primary graphical tools used in the detailed design phase were structure charts. Innovative uses of the structure charts were noted. In order to indicate a hardware function, a double-lined box was used, as shown below:



Because tasking was used extensively at the detailed design phase as well as at the design phase, tasks needed to be depicted on the structure charts too. The traditional structure chart does not provide for the drawing of concurrent processes; however, a successful representation is illustrated below:

n



The number n above the parallelogram indicates the number of copies of this task that exist in the system. Given that tasks are processes that go on concurrently with other processes, tasks that are spawned at the same level are indicated by placing as many of them on the same horizontal level. This technique avoids associating each task with a different level of decomposition, a result of the hierarchical implications of the vertical ordering of boxes on a structure chart. In the traditional structure chart, lines connecting boxes are arrowless -- the direction is by definition from top to bottom. In order to show the direction of intertask communications, arrowheads are put on lines; thus, if a box has a line connecting it to a task T, such that the arrow points to T, then T accepts an entry call from that box.

An additional point of interest in using traditional structure charts in an Ada environment lies in the fact that the traditional chart would imply single entry tasks. The arrow connecting two modules means that the module located higher on the page calls the module located lower on the page. The called module is subordinate to the calling module and exactly the same process occurs every time this module is called. Tasks, on the other hand, may have multiple entries, thus, the same code is not necessarily run each time an entry is called. Data that is passed between two modules in a structure chart is always the same for each call. Depending on which entry in a task is called, the data that is passed will be different. Therefore, the traditional modules depicted in a structure chart are not the most appropriate method to designate a task. The relationship between tasks and structure charts is complex and is discussed in more detail in the case study TASKS AND STRUCTURE CHARTS.

Structure charts are not the only graphical tools used in software development, and an area that needs more exploration is the relationship of tasks to these other tools. For instance, how would one model tasks using SADT, data flow diagrams, or system entity diagrams?

Dependency charts were introduced as a means of identifying what was imported by each compilation unit. These charts were maintained manually, but they were named an excellent candidate for automation. Their value was noted both in the development phase (to determine the order of compilation) and in maintenance (as a reference tool).

4.3 Ada as PDL

With respect to Ada's use as a PDL in the detailed design phase, there was some question as to the purpose of the PDL because of its similarity to the actual code. Too fine a decomposition on the structure charts was found to lessen the value of the PDL because it became too detailed (or else was simply redundant with the charts).

In the detailed design phase, several tasking issues arose. One of the areas discussed was the design of a function with dependent subfunctions which could operate concurrently. This led to a solution using dependent tasks, which is presented in the case study USE OF DEPENDENT TASKS. Another item of consideration at the detailed design level was the mechanism through which tasks can be suspended or stopped. Very often when a task must be stopped because it has reached the limit of its processing time, but must be subsequently restarted with a new set of data, what is in fact needed is a task suspension. This may be accomplished by preempting the task. There are several issues involved here and they are presented in the case study TASK PREEMPTION.

At higher levels of the design, queues and system behavior on queue underflow and overflow conditions may be specified; however, the actual queueing mechanism is not developed. At the detailed design level, this discussion becomes appropriate. In Ada, one may have either implicit or

explicit queues. Tasks provide implicit queues because entry calls are queued until the task is ready to accept the next one. There are applications where an explicit queue is needed, enabling the designer to select the next item to be processed on the basis of priority or to dequeue elements which have not been processed within a designated period of time. Queues are discussed further in the case study QUEUES AND GENERICS.

Embedded military computer systems use classified information. In developing these systems, the question arises as to how using Ada would affect the need to keep certain information classified. Some of this classified information is in the form of constants and other values, and in this form, there is a means of handling it in Ada. All classified values would be implemented as generic parameters to the system, thereby deferring their being given their real value until such time as the system is installed in the field. For the purposes of system test and integration, the system may be instantiated with a different set of values than the classified set.

The detailed design phase is the phase during which most packages are identified and specified. Declaring a package is largely based on intuition . Certain subprograms or tasks or data items seem to belong in a particular package. Guidelines are needed in this area, as are criteria to determine what constitutes a package.

4.4 Exceptions

It became clear in the course of the study that the use of exceptions is a serious design issue, and that guidelines and standards need to be developed governing their use. Even the developer of the most insignificant routine cannot ignore the issue of exceptions, for the following reasons:

1. In case of unknown and unexpected exceptions, the subprogram should clean up and re-raise the exception.

2. On the other hand, certain exceptions should be handled in a specific manner. For example, if the system user requests a certain operation and the operation raises STORAGE_ERROR, the man-machine software should simply tell the user that the operation cannot be carried out, and then continue normally.

3. Tight control needs to be exercised over the deliberate raising of exceptions. Some programmers might exhibit the tendency to use exceptions to "wash their hands" when they are not sure of what should be done under certain considerations (see also Section 5.5, Handling Impossible States).

In the study, no consensus was reached on what should be the proper use of exceptions. The following list of questions illustrates the problems involved.

1. Should predefined exceptions ever be raised explicitly? Stylistically it seems inappropriate; this practice would tend to defeat the purpose of built-in checks, since it would make it impossible to distinguish between totally unexpected errors and situations detected by the software.

2. When should one rely on built-in checks? It seems that certain practices of defensive programing will have a different flavor in Ada. For example, should range checks on input parameters be done explicitly, or should they be accomplished by having an appropriate subtype associated with the parameter?

   One guideline that can be suggested is to see whether the predefined exception or a different exception should be raised. In the canonical stack example, a negative count indicates stack underflow, and should probably raise a user-defined exception (say, STACK_UNDERFLOW) rather than the anonymous CONSTRAINT_ERROR. A second consideration is whether the exception is really indicative of an error. This aspect is considered further below.

3. Should "planned" exceptions be used for non-error situations? Consider an interactive program: it reads a line from the user, checks the correctness, and then performs some action. In checking the correctness, should user errors be handled by raising an exception (which, of course, would be handled at some level) or by returning a status code?

   The use of exceptions instead of status codes can lead to cleaner code; on the other hand, the programmer knows very well which exceptions will be propagated through each subprogram, and the simplicity of the code comes exactly from the programmer's understanding that each subprogram can propagate the exception without any clean-up. A maintainer might have a less complete understanding of the phenomenon, since the interconnections and mutual assumptions are implicit.

   In short, exceptions have some of the characteristics of global data (and, in fact, may be considered a form of side effect). They are undoubtedly a powerful design tool, but they are easy to misuse. A methodology for their proper usage needs to be developed.

4.5 Summary

Ada has been used during three phases of the life cycle so far. It is interesting to note that the language alone was not found to be sufficient to express the outputs of any one phase. In this particular phase Ada is used in conjunction with more traditional method of system development, structure charts. A certain amount of repetition was noted by the contractors between structure charts, low-level data flow diagrams and PDL.

4.6 Case Studies

The case studies illustrating this phase are:

- Tasks and structure charts
- Use of dependent tasks
- Task preemption
- Queues and generics

THIS PAGE INTENTIONALLY LEFT BLANK

TASKS AND STRUCTURE CHARTS

1. BACKGROUND

Objectives

To illustrate how different structure chart configurations can be modeled by Ada tasks or subprograms.

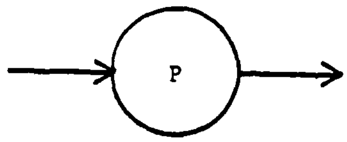
To illustrate how the same DFD (data flow diagram) node can result in different structure chart configurations.

To illustrate how the choice of connections in a structure chart maps directly into the structure of an Ada program.

To show how performance requirements affect design and how a design can be adapted to meet performance requirements.

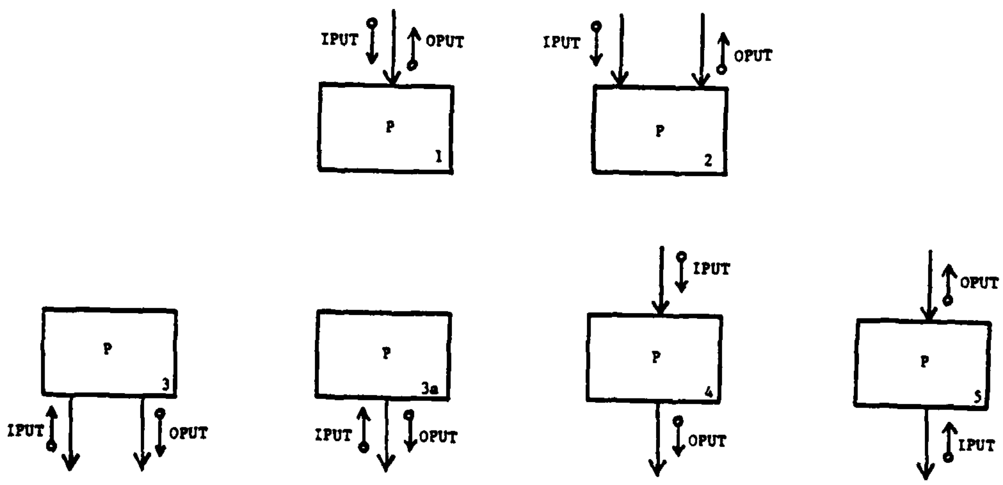
Designer's Problem

Given the following data flow structure, into what structure chart does it translate?



Discussion

The data flow node diagrammed above might result in any one of the following structure chart configurations



2. DETAILED EXAMPLE

Example Problem Statement

Each structure chart variation shown above will be discussed in the Detailed Solution section.

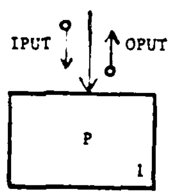
Solution Outline

Two approaches are discussed with each structure chart: the subprogram solution and the tasking solution.

Detailed Solution

One Call Accepting Input and Producing Output

Consider the following structure chart:



Subprograms

The structure shown above can be modeled by a procedure:

```
procedure P (IPUT: in T; OPUT : out U);
```

or a function:

```
function P (IPUT: in T) return U;
```

or, if the input and output are the same type:

```
procedure P(IPUT_OPUT: in out T);
```

An advantage of the subprogram approach is that P can be simultaneously invoked by more than one task. (Any subprogram can be executed concurrently if it is invoked by different tasks.) On the other hand, the caller of P must wait until OPUT is produced. The caller does not have the option of giving its input to P, going away to do other work then returning later to obtain the output.

If P accesses global data, P should not be visible to two or more tasks, since concurrent access to global data is error prone.

Tasks

The structure shown above can also be modeled by a task with a single entry:

```
task P is
     entry P(IPUT: in T; OPUT: out U);
end P;
```

or if T and U are the same:

```
task P is
     entry P(IPUT_OPUT: in out T);
end P;
```

Note that only one task can invoke P.P at one time -- no concurrency is possible while executing the entry.

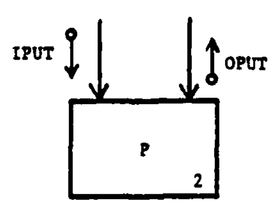
The caller must wait until task P accepts the call, and then the caller must wait until entry P.P produces its output and returns. In this sense, entry P.P is like a subprogram.

The main advantage of this use of a task as opposed to using a subprogram is that the task prevents simultaneous access to global statically allocated data, e.g.,

```
package R is
     task P is
          entry P (IPUT: in T; OPUT : out U);
     end P;
end R;

package body R is
     STATIC_LOCAL_DATA : ...;

     task body P is          -- accesses STATIC LOCAL DATA
          ...
     end P;
end R;
```

This approach guarantees that only one task at a time can access (i.e., read or modify) STATIC_LOCAL_DATA.

Accepts Input; Accepts Output Requests



Subprogram Approach

The structure above can be modeled by a package with two subprograms:

```
package P is
     procedure CONSUME (IPUT: in T);
     procedure PRODUCE (OPUT: out U);
end P;
```

This structure might be appropriate for storing and retrieving information in a database as long as the storing and retrieving operations are either not invoked simultaneously or the database is protected against simultaneous access by a task created within P's package body, e.g.,

```
package body P is
     task DATA_BASE is
          entry WRITE (IPUT: T);
          entry READ (OPUT: out U);
     end DATA_BASE;

     task body DATA_BASE is separate;

     procedure CONSUME (IPUT: T) is
     begin
          DATA_BASE.WRITE(IPUT);
     end;

     procedure PRODUCE (OPUT: out U) is
     begin
          DATA_BASE.READ(OPUT);
     end;
end DATA_BASE;
```

Tasks

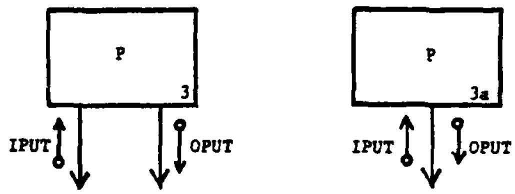
The structure shown above can also be modeled by the following task:

```
task P is
     entry CONSUME (IPUT: in T);
     entry PRODUCE (OPUT: out U);
end P;
```

This form is suitable when there is one task that is producing input (of type T) asynchronously, and another task needs to take the OUTPUT asynchronously, e.g., P might transform a stream of characters into validated messages. It is also the natural method for synchronizing access to a system-wide resource, as shown by the use of task shown above.

P might do no more than serve as a buffer, saving bursts of input until needed by the task calling PRODUCE. On the other hand, depending on how fast inputs arrive, how fast outputs can be produced, and how fast outputs are needed, the implementation of P can take different approaches. For example, P could queue its inputs, it outputs, or both. Various possibilities will be illustrated in the Epilogue.

Asks for Input; Disposes of Output



Subprogram Implementation

This structure can be modeled as a subprogram:

```
procedure P;
```

This subprogram could serve as the main program, for example.

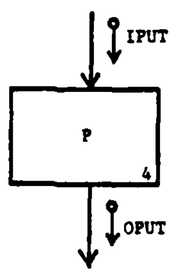
Task Implementation

The structure above can also be modeled by the following task specification:

```
task P;
```

There are no entries here because P calls other tasks or procedures to get its input and deliver its output. The only difference between 3 and 3a is that in 3a, a single call suffices to obtain input and deliver output, whereas in 3, separate calls are made for this purpose.

Accepts Input; Disposes of Output



Subprogram Implementation

This structure can be modeled by the following subprogram:

```
procedure P (IPUT: in T);
```

or it can be implemented as a package containing a procedure:

```
package P is
     procedure CONSUME (IPUT: in T);
end P;
```

The implementation as a package closely parallels the structure of the tasking implementation (see below). Note that the CONSUME procedure delivers its output by calling a lower level procedure (or task), and that CONSUME's caller must wait until CONSUME's output has been completely disposed of.

Tasking Implementation

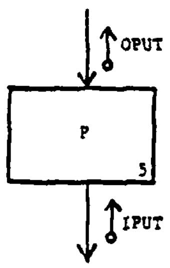
The structure above can also be modeled by the following task specification:

```
task P is
     entry CONSUME (IPUT: in T);
end P;
```

P delivers its output by calling a procedure or entry. CONSUME's caller need only wait until P has accepted its input. The caller can then continue executing in parallel with P. After P disposes of its output, it will be ready to accept more input.

Although P can queue its inputs, it is awkward for P to queue its outputs -- it is hard to check when the lower level entity needs an output.

Asks for Input; Accepts Output Requests



Subprogram or Package Implementation

The Ada implementation of this structure chart parallels that shown in the previous section.

```
        procedure P (OPUT: out U);
```

or

```
        package P is
             procedure PRODUCE (OPUT: out U);
        end P;
```

P (or PRODUCE) calls a lower level subprogram (or entry) when it needs more input. P's caller waits until output is ready.

Tasking Implementation

The following specification is analogous to the package specification given in the previous section.

```
        task P is
             entry PRODUCE (OPUT: out U);
        end P;
```

P calls some other procedure or task when it wants more input to process. It only makes sense for P to queue its outputs if it can guarantee that it won't be hung up trying to get more inputs.

3. EPILOGUE

In this section, two versions of a task that assembles sequences of characters into lines are presented. One of the structures illustrated in the previous section will be used to show the tradeoffs that are implied by the structure charts.

Specification of the Problem

We want task P to build a line of text from sequences of characters. The end of a line is signaled by an ASCII.LF; any ASCII.CR characters are to be ignored. We assume that the number of input characters seen before the LF sequence is in the range 0 .. MaxLine. We will use the TEXT data type as defined by the TEXT_HANDLER package in the Ada Reference Manual (Section 7.6).

Implementation With Server Task

Unbuffered Implementation

In this version, we assume the following task specification:

```
with TEXT_HANDLER;
package PKG is
     task P is
          entry Consume (Char: in CHARACTER);
          entry Produce (Line: out TEXT_HANDLER.TEXT);
     end P;
end PKG;
```

In the first version of the task body, after accepting a line of characters, we will accept no more input characters until the previous line has been taken away. (Note: with TEXT_HANDLER does not have to be repeated for the package body; see 10.1.1 of the Ada Reference Manual.)

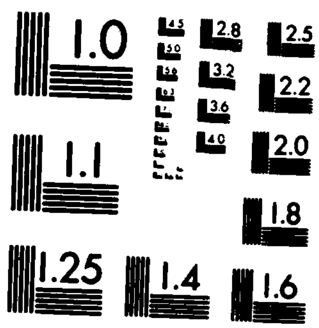
```
package body PKG is
     task body P is

          use TEXT_HANDLER;

          Line : TEXT(MaxLine);     -- line to be output
          CurChar : CHARACTER;      -- most recently accepted input
                                    -- character
     begin

          loop -- forever
               Set (Line, Value =>  ""); -- initialize value of Line

     BUILD_LINE:
               loop
     -- get character
                    accept Consume (Char : in CHARACTER) do
                         CurChar := Char;
                    end Consume;

     -- see if at the end of a line

               exit when CurChar = ASCII.LF;

                    if CurChar = ASCII.CR then
                         null;   -- ignore CR
                    else
                         Append (CurChar, To =>  Line);
                    end if;

               end loop BUILD_LINE;
```

AD-A124 996 ADA* SOFTWARE DESIGN METHODS FORMULATION CASE STUDIES REPORT(U) SOFTECH INC WALTHAM MA OCT 82 DAAK80-80-C-0187

2/3

UNCLASSIFIED

F/G 9/2



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
    --   deliver line
              accept Produce (Line : out TEXT) do
                   Set (Line, P.Line);
              end;
         end loop;
    end P;
end PKG;
```

In this solution, calls to CONSUME will be queued while waiting at the PRODUCE statement (and vice versa). Thus, the performance of CONSUME's callers is tightly coupled with the performance of PRODUCE's callers. To make this coupling looser, P's implementation can be modified to queue the assembled lines. This is the approach taken in the next solution.

Buffered Implementation

In the next version of the example, we will assume the use of the following generic package for creating FIFO queues:

```
generic
     type ELEM is limited private;
     SIZE: NATURAL;
with procedure ASSIGN (TO: in out ELEM; FROM: ELEM);
package FIFO_QUEUE is
     procedure PUT (InQueue : in ELEM);
     function Get return ELEM;
     function IsEmpty return BOOLEAN;
     function IsFull return BOOLEAN;
     OVERFLOW : exception;          -- raised by PUT
     UNDERFLOW: exception;          -- raised by GET
end FIFO_QUEUE;
```

(Since ELEM is limited private, assignment is not allowed. The SET generic parameter is used by FIFO_QUEUE to assign ELEMs to appropriate positions in a queue.) We can now produce the following task body, in which lines are queued for output. To ensure that an output line can be obtained at any time, we use a selective wait that will accept either a character or produce an output line if one is available in the queue.

```
with FIFO_QUEUE;
package body PKG is
     task body P is

          use TEXT_HANDLER;

          Line : Text(MaxLine);
          CurChar : CHARACTER;
          package QUEUE is new FIFO_QUEUE (TEXT(MaxLine), 50, SET);
```

```
    begin

         loop -- forever

              Set (Line, Value => "");

    BUILD_LINE:
              loop
    -- accept a character or deliver a line
                   select
                   when not QUEUE.IsEmpty =>
                        accept Produce(Line: out TEXT) do
                             SET (Line, Queue.Get);
                        end;
                   or
                        accept Consume (Char : in CHARACTER) do
                             CurChar := Char;
                        end;

    -- see if at end of line
              exit      when CurChar = ASCII.LF;
                        if CurChar = ASCII.CR then
                             null;     -- ignore CR
                        else
                             Append (CurChar, To => Line);
                        end if;
                   end select;
              end loop BUILD_LINE;

    -- if buffer is full, wait until there is room
              if QUEUE.IsFull then
                   accept Produce (Line : out TEXT) do
                        SET (Line, Queue.Get);
                   end;
              end if;

    -- otherwise, put line in queue
              QUEUE.Put (Line);
         end loop; -- get next line
    end P;
end PKG;
```

USE OF DEPENDENT TASKS

1. BACKGROUND

Case Study Objective

To illustrate how to use dependent tasks in Ada.

To illustrate use of exceptions raised in a rendezvous.

Designer's Problem

The general problem illustrated here is that two computations are to be executed concurrently; a third computation is started only after the first two have completed their processing. How can Ada tasking constructs be used to achieve the desired coordination among these computations?

2. DETAILED EXAMPLE

Example Problem Statement

The specific problem to be solved arose in a radar tracking system design (see Figure 1). A batch of radar returns (blips) is to be associated with tracking data maintained in a database. After a track is associated with a blip, track smoothing can be done concurrently with the process of finding the next track association. Predicting the next position for each track cannot, however, be done until all blips have been associated with tracks.

Solution Outline

The basic functionality being studied here is the target tracking mechanism. The design of the system specified that each radar sector would be allocated its own TRACK_TARGETS task to process all the detected blips in a given sector. (See Case Study TASK STRUCTURE FOR A TARGET TRACKING SYSTEM, Section 3.11, for a more detailed explanation of this processing.) Once the blips in a given sector have been identified, they must be associated with existing tracks. These tracks must then be smoothed (analogous to fitting a curve to the data points in the track). Lastly, the position of the next blip for each track must be predicted, using the associated and smoothed track data.

Given the above requirements, the designer must now determine what computations may go on concurrently, i.e. what should be modeled as tasks. The processing for each sector can go on concurrently with that for other sectors, and indeed this is implemented with tasking (TRACK_TARGETS). Because prediction needs the entire associated and smoothed track data set, it cannot operate in parallel either with the association or the smoothing computation. There is therefore no need to

implement the prediction computation as a task; a procedure is sufficient. Modeling this computation as a procedure does in fact introduce some parallelism because Ada procedures are reentrant. There are as many TRACK_TARGETS tasks as sectors, each one of which performs association, smoothing and prediction. Those computations modeled as procedures may be called from any of these tasks - they do not lock each other out (see the case study TASKS AND STRUCTURE CHARTS in Section 4.6 of this report). Thus prediction in different sectors can go on concurrently, even though in any particular sector, prediction cannot operate in parallel with association or smoothing.

Both the association and smoothing operations work on a track by track basis. So long as there is no data conflict, these can execute concurrently. Each track is associated once, then smoothed once. The smoothing can be done concurrently with the association by having it smooth the track which was just associated. Thus the smoothing computation can be implemented as a task and the association as a procedure. Smoothing rather than association is modeled as a task because if smoothing were a procedure, then even if association were a task, it would have to wait for the smoothing to be complete before proceeding with the next association. Association is a procedure just as prediction - there is no need to add any tasking complexity to its implementation.

A further point should be noted concerning the modeling of smoothing as a task. The requirements state that both association and smoothing must be completed before prediction begins. This can be enforced by making smoothing dependent on the ASSOCIATE_TRACK procedure. The Ada language states that the termination of any subprogram must await the termination of any dependent tasks, where dependent tasks include any tasks which are activated during the execution of the procedure in which they are declared or allocated (see Ada Language Reference Manual, Section 9.4). Thus ASSOCIATE_TRACK cannot return control to its caller, TRACK_TARGETS, until the task SMOOTH_TRACK completes execution. In other words, smoothing must be done for association to be done; implying that once association is done, prediction may start.

Detailed Solution

A structure chart for the first (dependent task) approach is shown in Figure 2. The Ada solution associated with this chart is given below.

```
procedure ASSOCIATE_TRACK (D: SECTOR_DATA) is
     task SMOOTH_TRACK is          -- dependent task
          entry DATA (TRACK : in TRACK_DATA);
     end;

     TRACK : TRACK_DATA;

     task body SMOOTH_TRACK is separate;
```

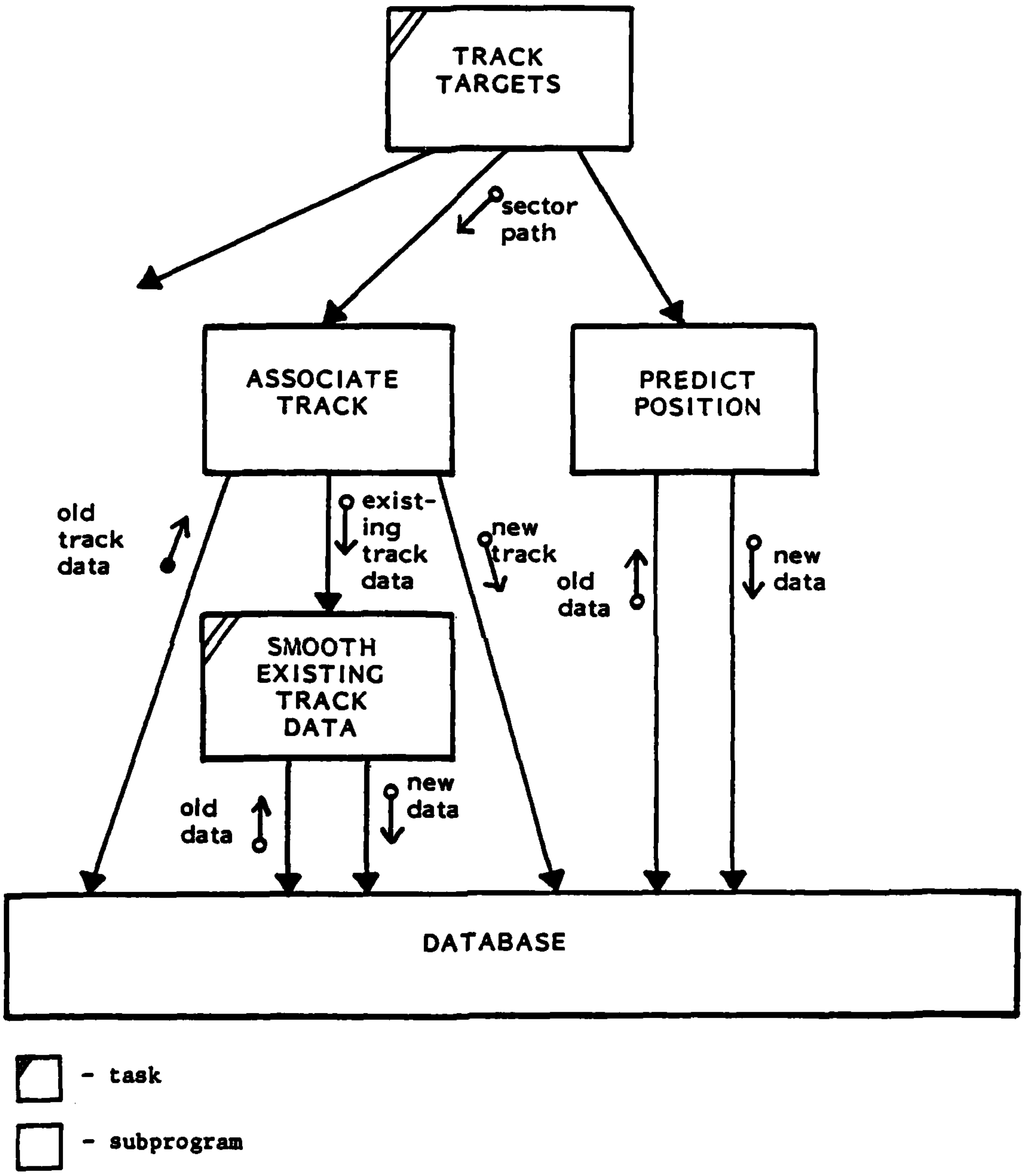


Figure 1. Structure Chart for Tracking Activity

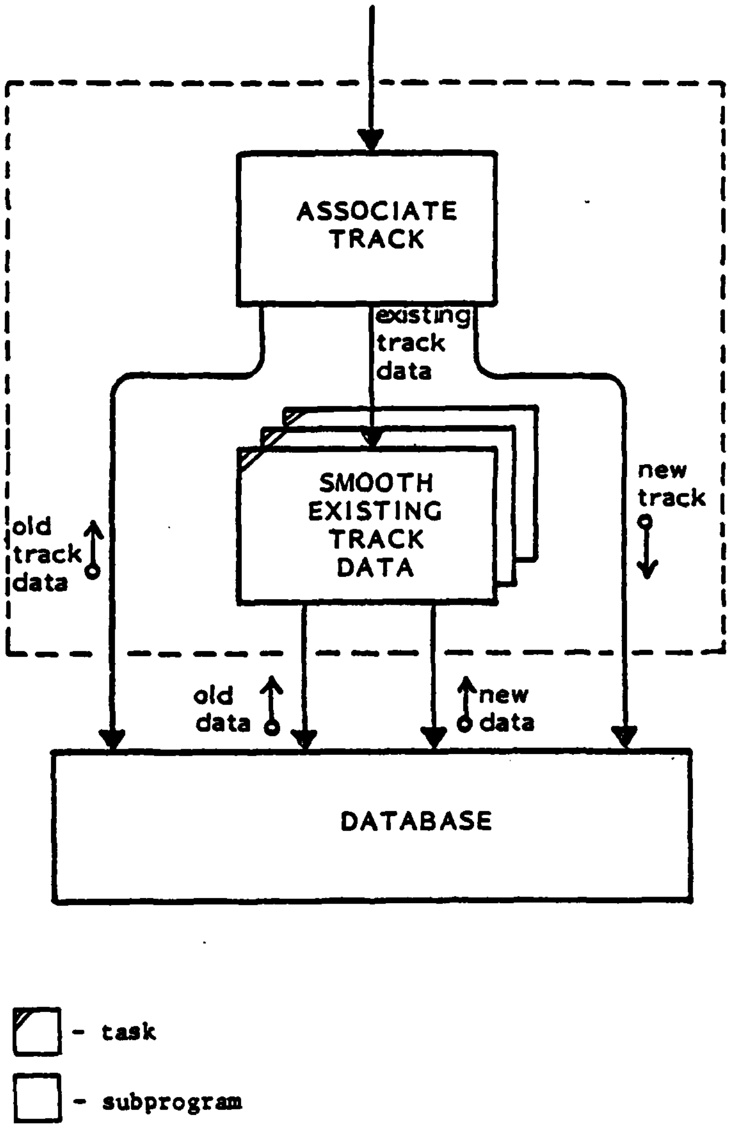


Figure 2. Structure Chart for Solution 1

```
begin
     for each blip in sector
     loop
          try to associate blip with track;
          if associated then
               SMOOTH_TRACK.DATA(TRACK);
          else
               initiate new track;
          end if;
     end loop;
end;      -- cannot leave until SMOOTH_TRACK terminates

separate (ASSOCIATE_TRACK)
task body SMOOTH_TRACK is
     TRACK : TRACK_DATA;
begin
     loop
          select
               accept DATA (TRACK : in TRACK_DATA) do
                    SMOOTH_TRACK.TRACK := DATA.TRACK;
               end;
          or
               terminate;
          end select;

          do track smoothing;
          update database;
     end loop
end SMOOTH_TRACK;
```

When ASSOCIATE_TRACK is at the end of its sequence of statements (and only then) the terminate alternative of SMOOTH_TRACK will be accepted, SMOOTH_TRACK will terminate, and control will return from ASSOCIATE_TRACK. Note that if smoothing takes more time than association (for each track), then it is possible to have more than one smoothing task. This only makes sense for an implementation if there are sufficient processing resources to execute concurrently all the smoothing tasks.

We will sketch this alternative by redefining ASSOCIATE_TRACK as a generic procedure with a parameter specifying the number of smoothing tasks to be used:

```
generic
     NUM_SMOOTHERS : POSITIVE;
procedure ASSOCIATE_TRACK (D : SECTOR_DATA) is
     task type SMOOTH_TRACK is
          entry DATA (TRACK : TRACK_DATA);
     end;
```

```
        SMOOTHERS : array (1..NUM_SMOOTHERS) of SMOOTH_TRACK;

        TRACK : TRACK_DATA;
        INDEX : POSITIVE range 1..NUM_SMOOTHERS := 1;

        task body SMOOTH_TRACK is separate;

    begin

        for each blip in sector
        loop
            try to associate blip with track;
            if associated then
            -- call least recently used smooth task
                SMOOTHERS(INDEX).DATA(TRACK);
                INDEX := (INDEX mod NUM_SMOOTHERS) + 1;
            else
                initiate new track;
            end if;
        end loop;
    end;
```

The code for SMOOTH_TRACK is as before.

Solution 2:

In this approach SMOOTH_TRACK is not a dependent task. The approach is much more complicated because SMOOTH_TRACK must call ASSOCIATE_TRACK to indicate when it is done smoothing the last track, and if there are no more tracks to be processed, then this fact must be communicated to SMOOTH_TRACK. Since SMOOTH_TRACK is to be a task, and since it is to call ASSOCIATE_TRACK, ASSOCIATE_TRACK must also be a task. (ASSOCIATE_TRACK cannot be a procedure because an <u>accept</u> statement can only appear within a task body, not a procedure.) The desired structure is indicated by the structure chart shown in Figure 3, but this structure chart cannot be realized in Ada since there is no way to give ASSOCIATE_TRACK two entries of which only one is visible to higher level modules. Hence, the more complex structure chart shown in Figure 4 is necessary, in which we create an interface procedure, ASSOCIATE_AND_SMOOTH_TRACK, and the task ASSOCIATE_TRACK is hidden within the module. Also, since we do not want dependent tasks, ASSOCIATE_TRACK and SMOOTH_TRACK must be tasks declared in library packages. Hence, we arrive at the following PDL solution:

```
    package ASSOCIATE is
        procedure ASSOCIATE_AND_SMOOTH_TRACK (D : SECTOR_DATA);
    end;
```

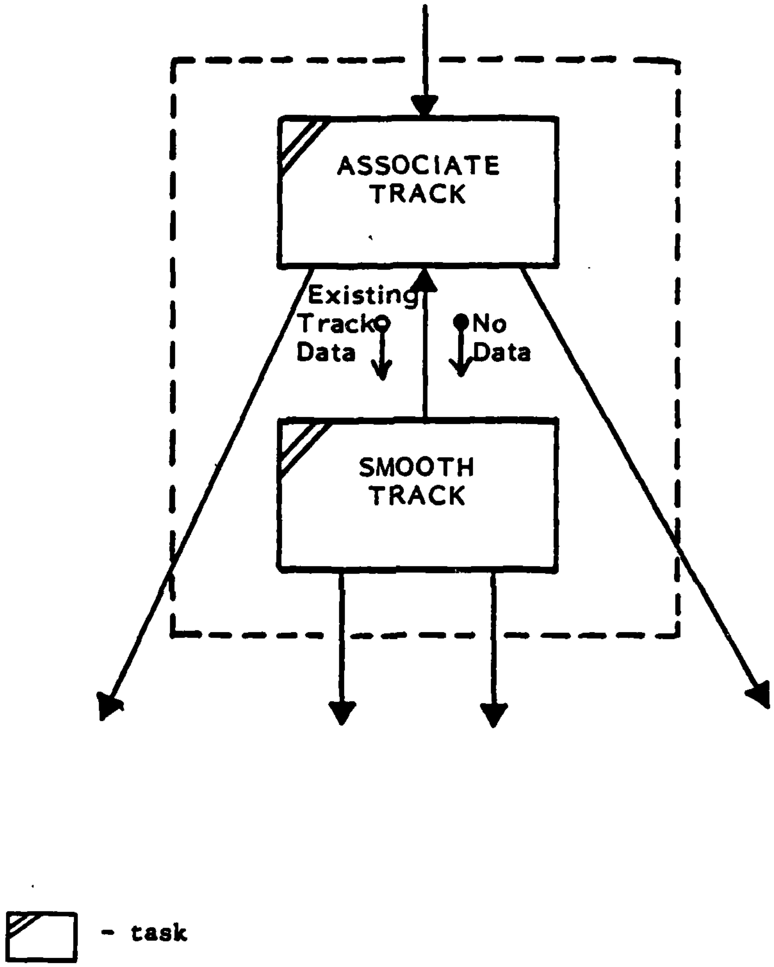


Figure 3. Initial Structure Chart for Solution 2

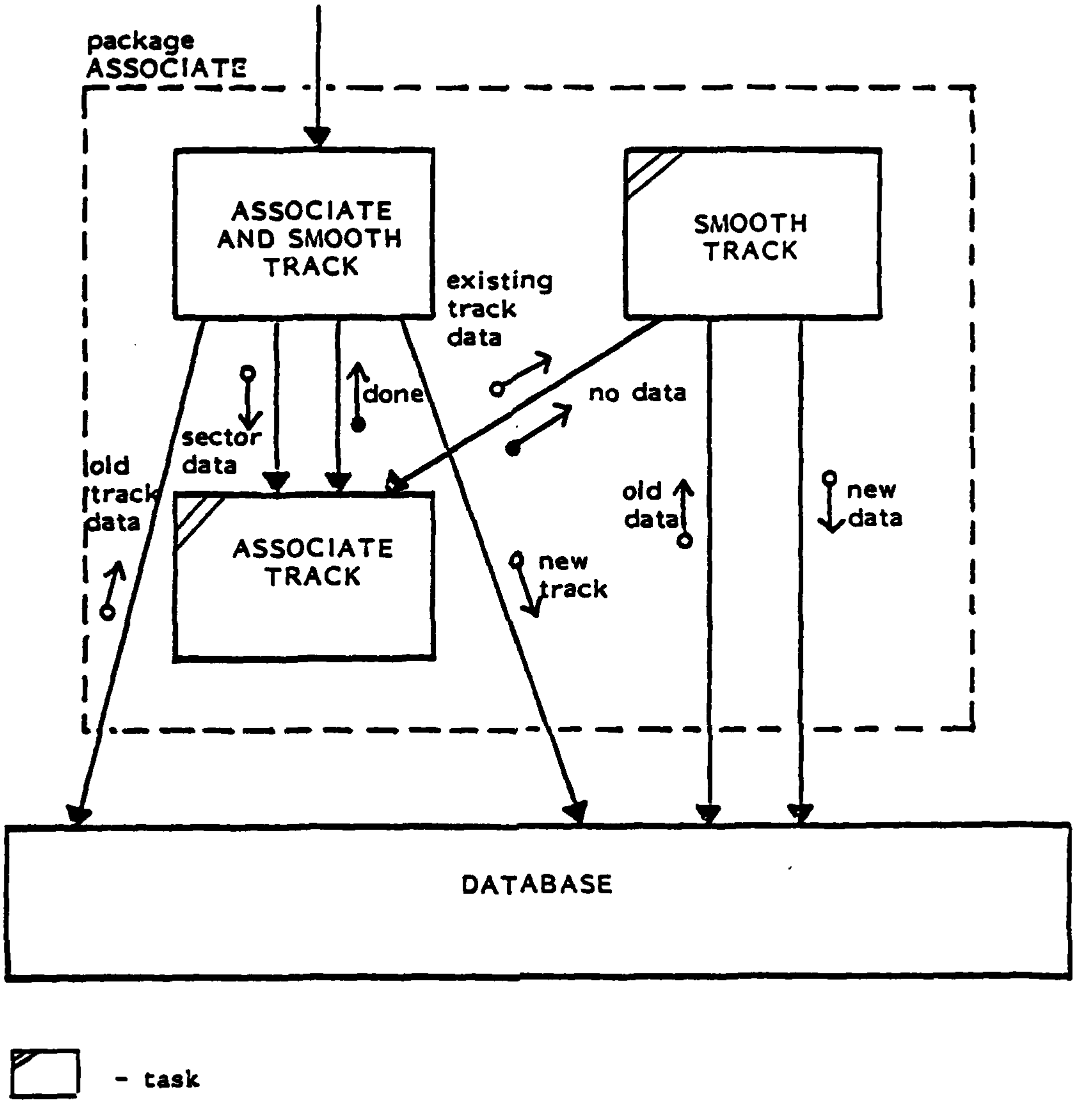


Figure 4. Final Structure Chart for Solution 2

```
package body ASSOCIATE is
     task ASSOCIATE_TRACK is
          entry DATA (D    : SECTOR_DATA);
          entry DATA (DATA : out TRACK_DATA);
          entry DONE;
     end;

     task SMOOTH;
     NO_DATA : exception; -- raised when all sector data processed

procedure ASSOCIATE_AND_SMOOTH_TRACK (D : SECTOR_DATA) is
begin
     ASSOCIATE_TRACK.DATA(D);
     ASSOCIATE_TRACK.DONE;    -- wait till done
end;

task body ASSOCIATE_TRACK is
     SECTOR : SECTOR_DATA;
begin
     loop
          accept DATA (D : SECTOR_DATA) do
               SECTOR := D;
          end;

          for each blip in sector
          loop
               try to associate blip with track;
               if associated then
                    accept DATA (DATA : out TRACK_DATA)
                         DATA := ...;
                    end;
               else
                    initiate new track;
               end if;
          end loop;

     -- need this block to tell when SMOOTH is done
          begin
               accept DATA (DATA : out TRACK_DATA) do
                    raise NO_DATA;
               end;
          exception
               when NO_DATA => null;
          end;

          accept DONE;
     end loop;
end ASSOCIATE_TRACK;
```

```
        task body SMOOTH is
             DATA : TRACK_DATA;
        begin
             loop
                  begin
                       ASSOCIATE_TRACK.DATA(DATA):  -- may raise NO_DATA
                       smooth track data;
                       update database;
                  exception
                       when NO_DATA => null;
                  end;
             end loop;
        end SMOOTH;
   end ASSOCIATE;
```

Note that when an exception is raised in a rendezvous, it is propagated both to the calling task and to the task containing the accept statement. Thus both SMOOTH and ASSOCIATE_TRACK are able to respond properly to the fact that there is no more data for smoothing.

3. EPILOGUE

This exercise shows two alternative approaches from which the designer may choose when developing a design that involves dependent tasks.

TASK PREEMPTION

BACKGROUND

Case Study Objective

To illustrate techniques for prematurely stopping (i.e., preempting) a task's activity so some other activity can be performed by the task.

Designer's Problem

It is sometimes the case in embedded systems that a task's current processing activity is no longer what the system requires. For example, a task sending a message may have to be interrupted and told to send a different, higher priority message. The proper way of stopping an Ada task so it can do something else is the subject of this case study.

Discussion

The essence of the preemption problem is that one task, the controlling task, needs to redirect the activities of another task, the preemptable task. In particular, it is necessary to make the preemptable task stop its current activity. An incorrect way of obtaining this effect occurs to programmers that are using Ada for the first time, namely, to abort the preemptable task. But aborting a task is a drastic action that should only be taken when a task has ceased to respond to any attempts to control it. Aborting a task routinely is likely to produce subtle bugs that depend on what a task is doing when it is aborted. For example, if a task is aborted while it is in a rendezvous to update a database, the database could be left in an inconsistent state.

Instead of aborting the preemptable task, the preemptable task should check periodically to see if it should continue its processing. There are basically two appropriate ways for the preemptable task to learn of a preemption request:

- via a STOP entry that is periodically checked to see if it has been called (see Case Study A):

```
select
     accept STOP;
     -- control comes here if preempted
else
     -- not preempted
end select;
```

  When the controlling task calls the STOP entry, the preemptable task terminates its current processing in an appropriate fashion.

- via a third task that, in essence, buffers preemption requests (see Case Study B). This preemption buffering task has two entries: TO_PREEMPT and IF_PREEMPTED. The preemptable task calls the IF_PREEMPTED entry to see if it should stop its processing. The controlling task calls the TO_PREEMPT entry to request preemption:

```
task body DECIDE is
begin
     loop
          accept TO_PREEMPT;
          accept IF_PREEMPTED;
     end loop
end DECIDE;
```

  The controlling task calls DECIDE.TO_PREEMPT; the preemptable task makes a conditional call to DECIDE.IF_PREEMPTED:

```
select
     DECIDE.IF_PREEMPTED;
     -- control goes here if preempted
else
     -- control comes here if not preempted
end select;
```

  Note that with this solution, a second TO_PREEMPT call will not be accepted until the first call has been acted upon, i.e., until the preemptable task has called DECIDE.IF_PREEMPTED. Now suppose the controlling task decides to preempt the preemptable task after the task has made its last check for preemption. If nothing is done, the next time the preemptable task is initiated, it will discover that it has been preempted as soon as it checks for preemption, but this preemption request is left over from its last use. To prevent this sort of thing from happening, the controlling task must ensure that the preemption state is properly set before it gives the preemptable task work to do. For the above implementation of DECIDE, it can ensure the task is in the not-preempted state by making the following conditional call:

```
select
     DECIDE.IF_PREEMPTED;
else
     null;
end select;
```

If a preemption request has been left over, the entry call will succeed, and this will reset the DECIDE task to the not-preempted state in which it is waiting for a TO_PREEMPT call. If no preemption request is left over, the DECIDE task is already in the not-preempted state, and will not be able to accept the IF_PREEMPTED call. The net effect is to guarantee that DECIDE is in the non-preempted state.

The use of conditional entry calls is somewhat cumbersome. An alternative implementation of DECIDE is to use a Boolean variable to store the preemption state:

```
task body DECIDE is
     preempted : BOOLEAN := FALSE;
begin
     loop
          select
               accept TO_PREEMPT do
                    preempted := TRUE;
               end TO_PREEMPT;
          or
               accept IF_PREEMPTED (STATUS : out BOOLEAN) do
                    STATUS := preempted;
                    preempted := FALSE;
               end IF_PREEMPTED;
          end select;
     end loop;
end DECIDE;
```

With this implementation, a call to TO_PREEMPT or IF_PREEMPTED will always be accepted almost immediately, and repeated calls to TO_PREEMPT will not cause problems.

It would be easier to use the above approach if IF_PREEMPTED were a function (named PREEMPTED) returning TRUE or FALSE. In addition, having the controlling task call PREEMPTED to initialize the state of the buffering task does not contribute to program clarity. As an exercise write a package called DECIDE that has the procedures TO_PREEMPT and NOT_PREEMPTED (to initialize the state) together with the function PREEMPTED. The bodies of these subprograms will call the appropriate entries of a preemption buffering task.

There are several points to consider when deciding which approach to use. Each approach requires dividing the preemptable task's processing into steps. The time to complete each step must be less than the maximum time allowed to respond to a preemption request. Preemption checks are performed between the steps. The first approach, accepting a STOP entry, is suitable if the number of steps is small and the steps divide

naturally into processing units of an appropriate length. Having natural step divisions is important because the accept statement for the STOP entry can only be written within a task body. It cannot be written inside a subprogram called from a task. Therefore, the general structure of the STOP entry approach is:

```
    task body ... is
    begin
         loop
              accept START_WORK;
              step 1;
              check STOP;
              if preempted then
                   preempted during step 1;
              else -- not preempted
                   step 2;
                   check STOP;
                   if preempted then
                        preempted during step 2;
                   else -- not preempted
                        ...
                   end if;
              end if;
              -- done work or preempted
         end loop;
    end;
```

Since the check STOP statement is actually implemented as a conditional wait, the actual code has the following structure:

```
    task body ... is
    begin
         loop
              accept START_WORK;
              step 1;
              select
                   accept STOP;
                   preempted during step 1;
              else -- not preempted
                   step 2;
                   select
                        accept STOP;
                        preempted during step 2;
                   else -- not preempted
                        ...
                   end select;
              end select;
         end loop;
    end;
```

If one of the steps is modified so it takes more time than originally expected, repartitioning the steps can be a complicated modification, or can make the structure of the algorithm more difficult to understand.

In short, this approach is not really desirable when the preemptable task has more than one step or might have to be divided into more than one step as requirements change. In addition, this approach is not usable if the controlling task cannot wait until its STOP call is accepted.

Providing an intermediate task has none of these disadvantages. The entry call (or function call) to check for preemption can be written in subprograms, as for any call. Hence, one needs to be less concerned with dividing the required processing into steps that make natural units. Instead, a call to check for preemption can be placed at any point where preemption can be handled conveniently. In addition, the controlling task does not wait until its preemption request is satisfied.

The intermediate task approach is not so convenient if the preemptable task must occasionally accept entry calls from other tasks during its preemptable processing. Accepting such calls requires a timed wait, since while waiting for the entry call the task must also check for preemption:

```
loop -- busy wait
     select
          accept MORE_WORK;
          do additional processing;
          exit;
     or
          delay 1.0;
          exit when DECIDE.PREEMPTED; -- preemption check
     end select;
end loop;
-- finished work or preempted
```

This mixture of checking for preemption and accepting entry calls is handled more efficiently if the STOP approach is used:

```
select
     accept MORE_WORK;
     do additional processing;
or
     accept STOP;
     -- clean up for preemption
end select;
```

Neither contractor discovered a case requiring the above solution.

The detailed case studies that follow illustrate both approaches to preemption.

THIS PAGE INTENTIONALLY LEFT BLANK

CASE STUDY A
TASK PREEMPTION

DETAILED EXAMPLE A

Example Problem Statement

A radar sweeps over n sectors in a single scan. After each sector is scanned, the blips detected in it must be processed before this sector is swept by the next radar scan. Each sector undergoes identical processing, and each sector can be processed asynchronously. Given these considerations, it has been decided to model the processing for each sector by a single task.

The problem may be stated in the following terms: given the task per sector representation, how does one express the requirement that the processing for a given sector must be completed in the time the radar takes to sweep n-1 sectors. If the processing is not completed in this time, it must be stopped. The processing task is then restarted with the fresh data from the new radar sweep of the sector.

Solution Outline

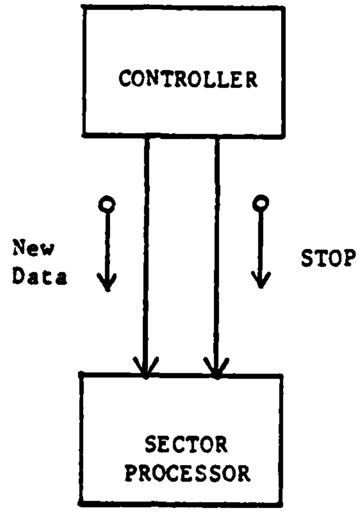
In order to stop the sector processing task after the radar has scanned n-1 sectors, one could abort that task. This solution, however, is dangerous because the task might be aborted at an awkward moment, e.g., while it is updating a database.

Upon closer analysis, an equivalent statement is as follows: after the radar has scanned n-1 sectors, the sector processing task is preempted and resumes when fresh data is available. A PDL statement of the above is straightforward:

```
loop
     get new data for sector;
     while blips to process and not preempted
     loop
          process a blip;
          check for preemption;
     end loop;
end loop;
```

One way to preempt the sector blip processing task is to use a selective wait whose alternatives are to accept a preemption command from the controlling task or to resume processing a blip. Care must be taken when using this technique to avoid creating a situation where either deadlock might occur or old data might be reprocessed. Both these possibilities are discussed in greater detail below.

The basic mechanism consists of a sector processing task with two entries. One entry allows this task to get the radar data and the other entry allows this task to be preempted should its processing time have run out. The structure chart is shown below:



An initial implementation of the PDL could look like this:

```
task type SECTOR_PROCESSING_TASK is
     entry NEW_DATA(S: ...);
     entry STOP;
end SECTOR_PROCESSING_TASK;

task body SECTOR_PROCESSING_TASK is
     LOCAL_DATA: ... ;
begin
     loop
          accept NEW_DATA(SECTOR_DATA: ... ) do
               LOCAL_DATA := SECTOR_DATA;
          end NEW_DATA;
          while BLIPS_TO_PROCESS
          loop
               PROCESS_BLIP;
               select                 -- If preempt call has occured
                    accept STOP;      -- then accept it, exit the blip
          -- exit loop                -- processing loop and be ready
                    exit;             -- to accept fresh data, or else
               else                   -- continue to loop through and
                    null;             -- process the available blip
               end select;
          end loop;
     end loop;
end SECTOR_PROCESSING_TASK;
```

The relevant part of the controller task would look like:

```
task body CONTROLLER is
    -- ...
    type SECTOR_PROCESSORS is array (SECTOR'FIRST .. SECTOR'LAST)
        of SECTOR_PROCESSING_TASK;
    SECTOR_PROCESSING:  SECTOR_PROCESSORS;
    -- ...
begin
    -- ...
    loop
        -- Give new data for this sector
        SECTOR_PROCESSING(CURRENT_SEC).NEW_DATA(SECTOR_DATA);
        -- Stop next sector's processing
        SECTOR_PROCESSING((CURRENT_SEC mod
                           SECTOR_PROCESSORS'LENGTH)+ 1).STOP;
                            -- Assume the array index starts at 1;
        accept NEW_SECTOR_PULSE;      -- Synchronize with the radar
    end loop;
    -- ...
end CONTROLLER;
```

An analysis of this solution reveals the following flaw. Consider the case that there is sufficient time to process all the blips, so that the sector processing task is now waiting to accept new data. The controlling task will issue its stop call after the radar has scanned n-1 sectors. The tasks are deadlocked because the sector processing task cannot proceed until it receives a NEW_DATA call, and the controller task cannot proceed until its STOP call is accepted.

One way to eliminate the deadlock is to put the NEW_DATA inside a selective wait whose other alternative is to accept a STOP call. The relevant part of this solution follows:

```
loop
     select
          accept NEW_DATA(SECTOR_DATA: ... ) do
               LOCAL_DATA := SECTOR_DATA;
          end NEW_DATA;
     or
          accept STOP;
     end select;
     while BLIPS_TO_PROCESS
     loop
          PROCESS_BLIP;
          select
               accept STOP;
     -- exit loop
               exit;
          else
               null;
          end select;
     end loop;
end loop;
```

This solution introduces a new problem, however, in that the old data gets processed anew, instead of the fresh data being received. Indeed one does not miss the STOP call, but as soon as the preemption is acknowledged, the flow of control drops through to the inner loop and the old data is reprocessed. The new data is not received until all the old data is reprocessed; the controller task hangs until the new data is accepted and in short, the entire processing goes haywire.

Both these problems can be avoided by introducing another control statement after the blip processing loop. If there are no more blips to process then the sector processing task waits until it receives a STOP call. This solution is shown in the following section. It is interesting to note that at the PDL level this extra control statement is not visible; it becomes necessary at the implementation level.

Detailed Solution

```
task SECTOR_PROCESSING_CONTROLLER is
    entry NEW_SECTOR_PULSE;
end SECTOR_PROCESSING_CONTROLLER;

task body SECTOR_PROCESSING_CONTROLLER is

    subtype BLIPS_ALLOWED is INTEGER range 0 .. MAX_BLIPS;
    type BLIPS(NUM_BLIPS : BLIPS_ALLOWED) is
         record
              DATA : BLIP_DATA(1..NUM_BLIPS);
         end record;

    task type SECTOR_PROCESSING_TASK is
         entry NEW_DATA(S : BLIPS);
         entry STOP;
    end SECTOR_PROCESSING_TASK;

    type SECTOR_PROCESSORS is array (SECTOR'FIRST .. SECTOR'LAST)
                             of SECTOR_PROCESSING_TASK;

    SECTOR_PROCESSING: SECTOR_PROCESSORS;

    task body SECTOR_PROCESSING_TASK is

         LOCAL_DATA: BLIPS;
         PREEMPTED : BOOLEAN;
         procedure PROCESS_BLIP(A_BLIP : BLIP_DATA_ELEMENT) is separate;

    begin
         loop
              accept NEW_DATA(SECTOR_DATA : BLIPS) do
                   LOCAL_DATA := SECTOR_DATA;
              end NEW_DATA;
              PREEMPTED := FALSE;
              for CURR_BLIP in 1 .. LOCAL_DATA.NUM_BLIPS
              loop
                   PROCESS_BLIP_LOCAL(DATA.DATA(CURR_BLIP));
                   select
                        accept STOP;
                        PREEMPTED := TRUE;
                   else
                        null;
                   end select;
                   exit when PREEMPTED;
              end loop;
              if not PREEMPTED then
                   accept STOP;
              end if;
         end loop;
    end SECTOR_PROCESSING_TASK;
```

```
begin    -- SECTOR_PROCESSING_CONTROLLER
    -- ...
    loop
        for CURRENT_SECTOR in SECTOR'RANGE
        loop
            accept NEW_SECTOR_PULSE;
            SECTOR_PROCESSING((CURRENT_SECTOR mod NUMBER_OF_SECTORS)+1).STOP;
                                -- Stop following sector's processing
            SECTOR_PROCESSING(CURRENT_SECTOR).NEW_DATA(SECTOR_DATA);
        end loop;
    -- ...
    end loop;
end SECTOR_PROCESSING_CONTROLLER;
```

EPILOGUE A

This exercise illustrates a way of suspending a task and letting it resume. Ada tasking does not provide for regularly starting and stopping tasks. For applications which seem to need such a capability, the designer should express this requirement using one of the tasking constructs discussed. As an exercise, consider the following situation:

- Assume that the time to process a blip is longer, and that the processing of a blip consists of two well-defined phases. Modify the design to check for preemption of the processing in each of the processing subphases.

CASE STUDY B
TASK PREEMPTION

DETAILED EXAMPLE B

Example Problem Statement

A message switch must be able to preempt the sending of a particular message when a higher priority message is to be sent to the same destination. The section of the message switch responsible for sending messages is organized as follows: for each logical output line emanating from the switch, a queueing task decides which message should be sent on a particular physical line (there could be more than one physical line per logical line). Each physical line is controlled by a task that actually transmits the message.

When the queueing task receives a message of higher priority than any message currently queued or being sent, the queueing task must preempt the physical line task that is sending a lower priority message, if no physical line is immediately available to send the message.

Solution Outline B

A structure chart showing the proposed solution is given in Figure 1a. A buffering task is used to handle preemption requests. However, the buffering task must do more than simply buffer preemption requests -- it must also validate the message headers before transmitting them to the sending task. In addition the queueing task must be notified when the physical line is available for another message. In the original design, LINE_READY was called by the buffering task. This caused a potential deadlock between the queueing and buffering task. The potential deadlock was resolved by having the buffering task alternate between accepting a preemption request and attempting to call LINE_READY. This required a busy wait. The solution proposed in this case study eliminates the busy wait and the potential deadlock.

The overall logic is as follows:

- when the queueing task determines that a message is to be sent and a physical line is free (LINE_READY), it calls the PUT_MESSAGE entry of the buffering task for that line, passing a pointer to the message to be sent.

- the buffering task validates the message header, and if the header is valid, gives the message to the sending task. If it is not valid, the sending task gets a null message, and notifies the queueing task that the line is ready.

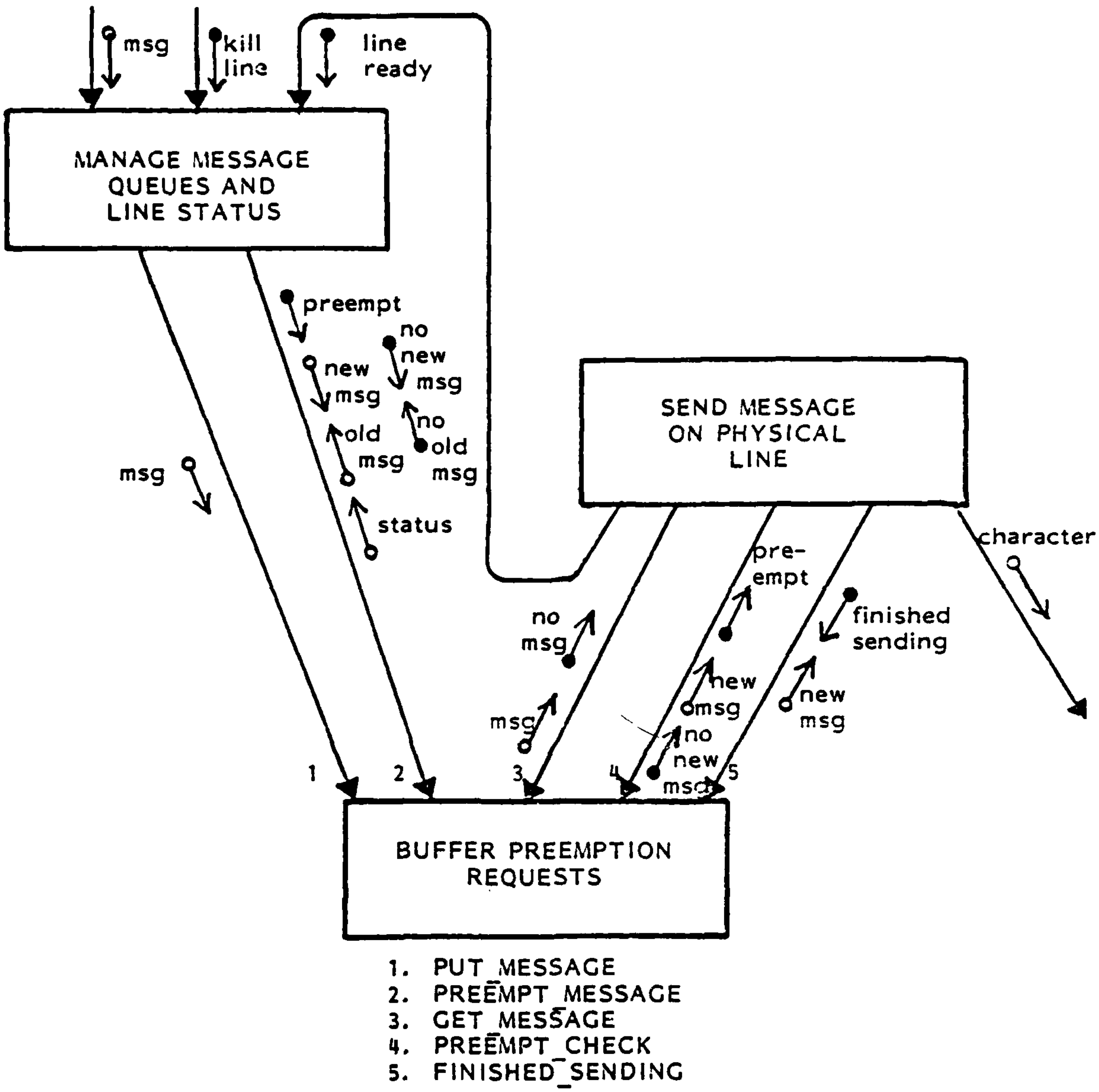


Figure 1a. Structure Chart for Output Message

- if the queueing task finds it necessary to preempt the sending task, the queueing task calls PREEMPT_MESSAGE with a new message to send. Once the buffering task has accepted a preemption request, the sending task must send it. The preempted message is returned to the queueing task.

- given a valid preemption request, the sending task will be allowed to call successfully PREEMPT_CHECK, thereby obtaining a new message to be sent.

- FINISHED_SENDING is called when the sending task has no more characters to send. Because the queueing task does not yet know that the sending task is finished, it might be attempting to preempt the sending task at the same time the sending task is finishing. For this reason, the FINISHED_SENDING call checks to see if a preempting message is ready to be sent. If so, it starts sending this message. If not, it will call LINE_READY of the queueing task.

- If an attempt is made to preempt the line after the sending task has called FINISHED_SENDING, but before it has called LINE_READY (of the queueing task), the preemption call will be denied, and the preempting message returned to the queue, which will very shortly discover that a line is available.

Once a message has been accepted by the buffering task, the queueing task considers the physical line to be busy. If the accepted message turns out to have an invalid header, the message cannot be sent. To ensure the line is returned to the ready state, the sending task is given a null message to send.

Figure 1b indicates what calls can be made at any given moment to the buffering task. This state diagram reflects the states of both the queueing and sending tasks. The buffering task starts out in state A. In this state, the sending task is idle, and the queueing task knows that the physical line is ready (LINE READY? YES), i.e., a message can be transmitted. A flag representing the buffering task's knowledge of the sending task's state indicates that the sending task is not considered to be sending a message (SENDING? NO).

When in state A, the queueing task will only call the buffering task using the PUT_MESSAGE entry. (This is indicated by the number 1 in the circle representing state A. The "1" corresponds to the numbering in Figure 1a.) In addition, the sending task is waiting for the buffering task to accept its call to GET_MESSAGE. The buffering task will not accept this call until the PUT_MESSAGE call has been accepted. This is indicated by showing that in state A, PUT_MESSAGE is the only call that can be accepted.

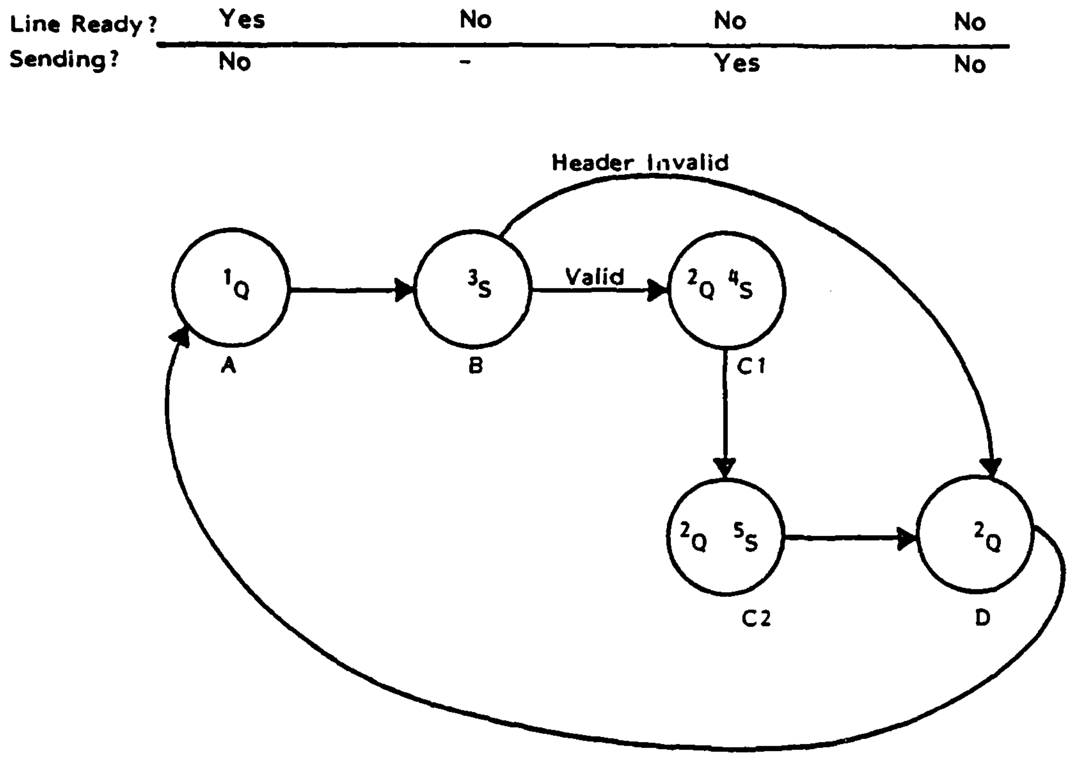


Figure 1b. Buffer Task States, Showing What Calls Can Be Received in Each State

When the PUT_MESSAGE call is received, the system transitions to state B. In this state, the queueing task has marked the line as not ready (LINE READY? NO), and the buffering task accepts the GET_MESSAGE call from the sending task. If the header of the message is not valid, a null message is given to the sending task, and the system transitions to state D. In this state, the queueing task can still issue a preemption call to the buffering task, since the queueing task believes the line is not available (i.e., a message is being sent). The sending task is given a null message so it will call the LINE_READY entry of the queueing task, telling the queueing task that no message is being transmitted. (Note: the procedure for validating the header tells the queueing task if the header is not valid, and in such a case, the queueing task disposes of the message appropriately.) The call to LINE_READY causes the system to transition back to state A.

If the header is valid, the system transitions to state C. In this state, the sending task is transmitting the message. The system is in state C1 until the sending task has finished transmitting the message. While in state C1, the buffering task can receive either a preemption request from the queueing task or a preemption check from the sending task.

When the sending task has finished, it attempts to call the FINISHED_SENDING entry of the buffering task; the attempt to call FINISHED_SENDING marks the transition to state C2. (Note that while the sending task is trying to call FINISHED_SENDING, the queueing task could be accepting a PREEMPT_MESSAGE call.) When the FINISHED_SENDING call is accepted the system transitions to state D and remains in this state until the sending task has notified the queueing task that the line is ready. When the system is in state D, the queueing task can still call PREEMPT_MESSAGE since the queueing task does not yet know that the sending task has finished.

This state diagram illustrates when it is possible for different entries of the buffering task to be called. Note that the LINE-READY entry of the sending task can only be called when the system is in state D, because this is the only state when the line is not considered ready by the queueing task even though the sending task has finished sending. If LINE_READY were called directly from the buffering task, the queueing task could be simultaneously trying to call the buffering task. Deadlock would result, since the queueing task cannot accept the LINE_READY call at the same time it is calling PREEMPT_MESSAGE. Deadlock is avoided here by placing the LINE_READY call in a third task (the sending task) and by ensuring that although the PREEMPT_MESSAGE call is accepted in state D, the queueing task is notified that preemption is not possible. By returning the preemption request, the queueing task is made able to accept the LINE_READY call, and deadlock is avoided.

PDL for the buffering task body is given in Figure 2. Actual code is shown in Figure 3. PDL for the sending task is given in Figure 4; the code is in Figure 5.

```
task body BUFFER_PREEMPTION is
begin
  loop
    select
      accept message from queueing task;   -- state A

      validate message header;             -- state B
      if not valid header then
        give null message to send task;
        sending := false;
      else    -- header is valid
        give valid message to send task;
        sending := true;
      end if;
    or
      accept preemption by queueing task do -- state C, D
        if not sending then                 -- state D
          if preemption message exists then
            return message to queueing task;
          else      -- ignore preemption request
            null;
          end if;
        else    -- currently sending a message   -- state C
          if preempting message exists then
            validate message header;
            if not valid then
              reject message and preemption request;
            else    -- header is valid
              wait for send task to finish or check for preemption;
              if finished sending then     -- state C2; preempted with new message
                return null message to queueing task;
                give new message to send task;
              else    -- send is not finished and is checking for preemption
                return current message to queueing task; -- state C1; preempted with new message
                give new message to send task;
              end if;   -- finished sending
            end if;   -- not valid
          else    -- no preempting message to send
            wait for send task to finish or check for preemption;
            if finished sending then       -- state C2; preempted with null message
              return null message to queueing task;
              give null message to send task;
              sending := false;
            else    -- send is to be preempted   -- state C1; preempted with null message
              return current message to queueing task;
              give null message to send task;
            end if;   -- finished sending
          end if;   -- preempting message present
        end if;   -- not sending
      end;    -- preemption by queueing task
    or
      accept finished sending from send task do  -- state C2, no preemption pending
        give null message to send task;
        sending := false;
      end;    -- finished sending from send task
    end select;
  end loop;
end BUFFER_PREEMPTION;
```

Figure 2. PDL for Buffering Task Body

```
task body BUFFER_PREEMPTION is
  Cur_Msg : MsgID;
  Sending : Boolean;
begin
  loop  -- forever
    select
      accept Put_Message (Msg : MsgID) do     -- only called when not sending
        Cur_Msg := Msg;
      end Put_Message;

      Sending := True;

      if not Header_Valid (Cur_Msg) then
        Cur_Msg := null;
        Sending := False;
      end if;   --  Header_Valid

      accept Get_Message (Msg : out MsgID) do
        Msg := Cur_Msg;   -- may be null
      end Get_Message;
    or
      accept Preempt_Message (New_Msg : MsgID;
                              Old_Msg : out MsgID;
                              Status  : out Msg_Stat) do
        if not Sending then
          Old_Msg := New_Msg;
          Status := Refused_Preemption;
        else    -- currently sending a message
          if New_Msg /= null then
            if Header_Valid (New_Msg) then
              select
                accept Preempt_Check (Msg : out MsgID) do
                  Old_Msg := Cur_Msg;
                  Cur_Msg := New_Msg;
                  Msg := Cur_Msg;
                end Preempt_Check;
              or
                accept Finished_Sending (Msg : out MsgID) do
                  Old_Msg := null;
                  Cur_Msg := New_Msg;
                  Msg := Cur_Msg;
                end Finished_Sending;
              end select;
```

Figure 3. Code for Buffering Task Body

```
              else    -- header not valid
                Old_Msg := null;
                Status := Rejected;
              end if;   -- Header_Valid
            else    -- no preempting message
              select
                accept Preempt_Check (Msg : out MsgID) do
                  Old_Msg := Cur_Msg;
                  Msg := null;
                  Status := Normal_ACC;
                end Preempt_Check;
              or
                accept Finished_Sending (Msg : out Msg_ID) do
                  Old_Msg := null;
                  Msg := null;
                  Status := Completed_ACC;
                end Finished_Sending;
              end select;
            end if;   --  New_Msg /= null
          end if;   -- not Sending
        end Preempt_Message;
      or
        accept Finished_Sending (Msg : out MsgID) do
          Msg := null;
          Sending := False;
        end Finished_Sending;
      end select;
    end loop;
  end BUFFER_PREEMPTION;
```

Figure 3. Code for Buffering Task Body (Continued)

```
task body Send_Task is
begin
  loop    -- forever
    call Get_Message for this physical line;

START_MSG_TRANSMISSION:
    while there is a message to send
    loop
      send transmission identification;
      log start of message transmission;
      status := ok;
      preempted := false;

SEND_MSG:
      while status = ok and not preempted
      loop
        preempted := false;
        get next block of message to send (status);

SEND_CHAR:
        for each character in message block
        loop
          check for preemption;
          if preempted then
            cancel transmission;
            log transmission cancellation;
            if no preempting message then
              call Finished_Sending of buffer task;
            end if;
            exit SEND_CHAR;
          else    -- not preempted
            send character;
          end if;
        end loop SEND_CHAR;  -- for each character in message block
      end loop SEND_MSG;  -- while status = ok and not preempted

      if not preempted then
        if status /= end_of_text then
          cancel message transmission;
          log transmission cancellation;
          send message to operator;
        else    -- status = end_of_text
          send end of message sequence;
          log end of message transmission;
          increment count of sent messages;
          send trailer sequence, if necessary;
        end if;   -- status /= end_of_text

        call Finished_Sending entry of buffering task;
        -- Finished_Sending may produce a non-null message to transmit
      end if;   -- not preempted
    end loop START_MSG_TRANSMISSION;   -- while there is a message to send

    tell queueing task that line is ready;

  end loop;   -- forever
end Send_Task;
```

Figure 4. PDL For Send_Task Body

```
task body Send_Task is

     New_Msg, Cur_Msg : MsgID;
     Phys_Line        : Physical_line;
     Preempted        : Boolean := False;
     SF_SF            : Boolean;

begin
     accept Initialize (Line : Physical_Line) do
          Phys_Line := Line;
     end Initialize;

     SF_SF := Line_Tbl_Ops.Spec_Term_Type'(Interswitch) =
                 Line_Tbl_Ops.Read_Spec_Term (Phys_Line);

     Notify_Operator ((Startup, Phys_Line));

     declare
          Buffer_Preempts : Buffer_Preemption renames
                                 Tasks.Table(Phys_Line).Output;
     begin
          loop  -- forever
               Buffer_Preempts.Get_Message (Cur_Msg);

START_MSG_TRANSMISSION:
               while Cur_Msg /= null
               loop
                    Buffer (1..TI_Lgth) := Generate_Transmission_ID (
                                             Read_Char_Set(Cur_Msg));
                    for I in 1 .. TI_Lgth
                    loop
                         Trans.Mit (Buffer(I));
                    end loop;

                    Log (SOM_Seq, Phys_Line, Cur_Msg);

                    Open_for_Read (Cur_Msg, Line_Tbl_Ops.Read_Start_Part);

                    Status := OK;
                    Preempted := False;

SEND_MSG:           while Status = OK and not Preempted
                    loop
                         Read_Continuous (..., Status);

SEND_CHARS:              for I in 1.. Buffer_Length
                         loop
                              select  -- check for preemption
                                   Buffer_Preempts.Preempt_Check (New_Msg);

                                   Send_Cancel_Transmission_Sequence;
                                   Log (Cantran, Phys_Line, Cur_Msg);
                                   Cur_Msg := New_Msg;
                                   Preempted := True;
                                   exit SEND_CHARS;
                              else
                                   Trans.Mit (Buffer(I));
                              end select;
                         end loop SEND_CHARS;  -- for I in 1..Buffer_Length
                    end loop SEND_MSG;  -- while Status = OK and not Preempted
```

Figure 5. Code for Send_Task Body

```
            if not Preempted then
                 if Status /= End_of_Text then
                      Send_Cantran_Sequence;
                      Log (Cantran, Phys_Line, Cur_Msg);
                      Generate_SVC_Message (Cur_Msg, Suspended_Transmission);
                      Notify_Operator (...);
                 else  -- Status = End_of_Text
                      Buffer (1..EOM_Lgth) := EOM_Seq;

                      for I in 1 .. EOM_Lgth
                      loop
                           Trans.Mit (Buffer(I));
                      end loop;  -- for I in 1 .. EOM_Lgth

                      Log (EOM_Out, Phys_Line, Cur_Msg);
                      Increment_OCSN (Phys_Line);

                      if not SF_SF then
                           Send_Trailer;
                      end if;
                 end if;  -- Status /= End_of_Text
            end if;  -- not Preempted

            Buffer_Preempts.Finished_Sending (Cur_Msg);
       end loop START_MSG_TRANSMISSION;  -- while Cur_Msg /= null

       Queueing_Task.Line_Ready (Phys_Line);
  end loop;  -- forever
 end;  -- block
end Send_Task;
```

Figure 5. Code for Send_Task Body (Continued)

THIS PAGE INTENTIONALLY LEFT BLANK

QUEUES AND GENERICS

1. BACKGROUND

Case Study Objective

To illustrate a generic queue.

Designer's Problem

The designer's problem is to decouple the interaction among several primary tasks. How do you design the intertask communication so that none of the primary tasks have to wait to receive or to transmit it?

Discussion

In order to decouple the tasks, a queue must be introduced to handle the data communication. There are two issues of interest here: 1) the representation (not to be confused with the implementation) of the queue; and 2) the choice of queueing discipline (e.g., blocking, nonblocking, FIFO, LIFO, etc.).

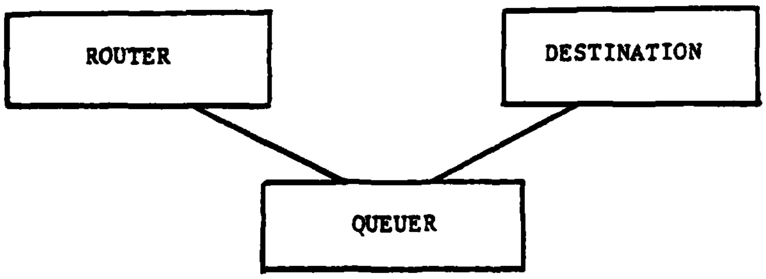
2. DETAILED EXAMPLE

Example Problem Statement

Within a larger system, there is a process which receives internal messages, analyzes their type, and routes them accordingly. The process which consumes these internal messages cannot afford to wait until the destination process is available to receive the message. Similarly, none of the destination processes can wait for the router to be free to find out if they have a message to process.

Solution Outline

In order to queue data flowing between two or more tasks, the queue must be represented as another task. Within this task are found the data structures to control the flow of items being passed. The router and destination processes can execute independently of each other because they do not need to communicate directly with each other. The queueing task handles the communication, as illustrated below:



Because of the QUEUER's implementation as a task, it has the ability to synchronize either with the router to receive a message or with the destination process to deliver a message. In other words, the queueing and dequeueing operations are implemented as entry calls.

This particular solution implements a first-in-first-out blocking queue: when the queue is full, it refuses to accept any more messages until the oldest message has been dequeued by a request from the destination process.

Queueing is an operation which is frequently used by the system designer. It therefore makes sense to offer the designer a reusable set of queueing packages, each of which provides a different queueing discipline. In order to achieve this objective, the blocking queue is presented as a generic package. The designer may then instantiate it for each blocking queue he needs. The generic parameters provided are size and message type.

Detailed Solution

```
generic
    Q_SIZE:  INTEGER;
    type MESSAGE_TYPE is private;
package BLOCKING_QUEUE is
    procedure PUT(X:  in MESSAGE_TYPE);
    procedure GET(X:  out MESSAGE_TYPE);
end BLOCKING_QUEUE;

package body BLOCKING_QUEUE is

    task WORKER is
         entry PUT(X:  in MESSAGE_TYPE);
         entry GET(X:  out MESSAGE_TYPE);
    end WORKER;

    procedure PUT(X:  in MESSAGE_TYPE) is
    begin
         WORKER.PUT(X);
    end PUT;

    procedure GET(X:  out MESSAGE_TYPE) is
    begin
         WORKER.GET(X);
    end GET;
```

```
task body WORKER is
     type Q_POINTER is range 0..Q_SIZE;
     type Q_TYPE is array (Q_POINTER range 1..Q_SIZE) of MESSAGE_TYPE;
     FIRST      :  Q_POINTER := 1;
     LAST       :  Q_POINTER := 0;
     CUR_SIZE   :  Q_POINTER := 0;
     QUEUE      :  Q_TYPE;
begin
     loop            -- forever
          select
               -- test whether there is room to queue more messages
               when CUR_SIZE < Q_SIZE =>
                    accept PUT(X:  in MESSAGE_TYPE) do
                         LAST := (LAST mod Q_SIZE) + 1;
                         QUEUE(LAST) := X;
                    end PUT;
                    CUR_SIZE := CUR_SIZE + 1;
          or
               -- test whether there are messages that could be
               -- dequeued
               when CUR_SIZE > 0 =>
                    accept GET(X:  out MESSAGE_TYPE) do
                         X := QUEUE(FIRST);
                    end GET;
                    CUR_SIZE := CUR_SIZE - 1;
                    FIRST := (FIRST mod Q_SIZE) + 1;
          end select;
     end loop;
end WORKER;

begin -- BLOCKING_QUEUE

     ...

end BLOCKING_QUEUE;
```

```
-- to create a blocking queue which holds up to 50 messages:

package MESSAGE_TYPE_DEFINITION is
    type MESSAGE is (SERVICE, INIT, LINK, FAIL);
    type SEVERITY is (SAFE, CAUTION_ADVISED, DANGEROUS, DISASTROUS);
    type MSG(MSG_TYPE:  MESSAGE) is
         record
              ID        :  INTEGER range 1..100;
              TEXT      :  STRING(1..200);
              case MSG_TYPE is
                   when SERVICE   =>
                        UNIT_NUM  :  INTEGER range 4..8;
                   when INIT      =>
                        TIME      :  DATE;
                   when LINK      =>
                        LINK_FROM :  INTEGER range 4..8;
                        LINK_TO   :  INTEGER range 4..8;
                   when FAIL      =>
                        DEGREE    :  SEVERITY;
              end case;
         end record;
    subtype S_MSG is MSG(SERVICE);
    subtype I_MSG is MSG(INIT);
    subtype L_MSG is MSG(LINK);
    subtype F_MSG is MSG(FAIL);
end MESSAGE_TYPE_DEFINITION;

with MESSAGE_TYPE_DEFINITION; use MESSAGE_TYPE_DEFINITION;
package SERVICE_MESSAGE_QUEUE is new BLOCKING_QUEUE(Q_SIZE      => 50;
                                                 MESSAGE_TYPE   => S_MSG);

with MESSAGE_TYPE_DEFINITION; use MESSAGE_TYPE_DEFINITION;
package INIT_MESSAGE_QUEUE is new BLOCKING_QUEUE(Q_SIZE         => 50,
                                                 MESSAGE_TYPE   => I_MSG);

with MESSAGE_TYPE_DEFINITION; use MESSAGE_TYPE_DEFINITION;
package LINK_MESSAGE_QUEUE is new BLOCKING_QUEUE(Q_SIZE         => 50,
                                                 MESSAGE_TYPE   => L_MSG);

with MESSAGE_TYPE_DEFINITION; use MESSAGE_TYPE_DEFINITION;
package FAIL_MESSAGE_QUEUE is new BLOCKING_QUEUE(Q_SIZE         => 50,
                                                 MESSAGE_TYPE   => F_MSG);
```

```
-- the router recognizes the message and queues it to the appropriate queue:

with MESSAGE_TYPE_DEFINITION; use MESSAGE_TYPE_DEFINITION;
with SERVICE_MESSAGE_QUEUE; use SERVICE_MESSAGE_QUEUE;
with INIT_MESSAGE_QUEUE; use INIT_MESSAGE_QUEUE;
with LINK_MESSAGE_QUEUE; use LINK_MESSAGE_QUEUE;
with FAIL_MESSAGE_QUEUE; use FAIL_MESSAGE_QUEUE;
procedure ROUTER(M: MSG) is
begin
    -- some processing
    case M.MSG_TYPE is
         when SERVICE   =>
              SERVICE_MESSAGE_QUEUE.PUT(M);
         when INIT      =>
              INIT_MESSAGE_QUEUE.PUT(M);
         when LINK      =>
              LINK_MESSAGE_QUEUE.PUT(M);
         when FAIL      =>
              FAIL_MESSAGE_QUEUE.PUT(M);
    end case;
    -- more processing
end ROUTER;

-- the destination process wishes to process service messages:

with MESSAGE_TYPE_DEFINITION, SERVICE_MESSAGE_QUEUE;
use MESSAGE_TYPE_DEFINITION, SERVICE_MESSAGE_QUEUE;
package SERVICE_MESSAGES is
    task PROCESS_SERVICE_MESSAGES;
end SERVICE_MESSAGES;

package body SERVICE_MESSAGES is
    task body PROCESS_SERVICE_MESSAGES is
         M: S_MSG;
    begin
         loop                 -- forever
              SERVICE_MESSAGE_QUEUE.GET(M);
         -- do more processing
         end loop;
    end PROCESS_SERVICE_MESSAGES;
end SERVICE_MESSAGES;

-- more queue servicing processes
```

3. EPILOGUE

Queues are essential to applications in Ada, and every designer should keep the following two things in mind. Firstly, he should know how to implement a queue. Secondly, having realized that writing a queue package is straightforward and requires only a small and predictable amount of time to do, he should consider queues as primitives from the top down design point of view. Thus he should proceed with the design, using queues like other Ada constructs, without worrying about the implementation details of queues.

This exercise illustrates a good use of generics because the functionality of a particular type of queue remains the same, regardless of the size of the queue and of the type of data that is being queued. Thus by writing one template, the designer has effectively created almost unlimited queueing power.

This example illustrates one kind of queue; the variety of queueing disciplines provides several subjects for classroom discussion and assignments, listed below:

- This example presented a blocking queue. Implement a nonblocking queue, taking specific action when an underflow or overflow situation is encountered. There are several ways of handling these potential error situations:

  - raising and handling an exception

  - discarding the oldest entry in a full queue to accommodate the newest entry

  - discarding the newest entry in a full queue, effectively ignoring any queueing requests until a dequeueing request has been received.

- It is sometimes desirable to have unbounded queues. These are implemented using allocators. As an exercise, they can be implemented, for both blocking and nonblocking queues.

- For a related but more difficult problem, implement prioritized queues.

Section 5

CODE/UNIT TEST PHASE

5.1 Purpose

The implementation of a design occurs in the code/unit test phase. In using Ada to write executable code, there are two areas of concern: the readability of the code and the best way of expressing an operation in Ada. This section discusses the problem of readability, including case studies on various aspects of readability, as well as numerous Ada coding paradigms for which a need was identified in the course of the technical interchange forums.

5.2 Readability

As with most code, Ada is read far more often than it is written. For the benefit of developers and maintenance programmers, Ada code should be as readable as possible. Starting with the structural element, the package, the question arises as to which is more readable, a single package with a hierarchy of nested packages inside of it, or many packages maintained as library units whose dependencies are indicated by with clauses.

On a smaller scale, the problem exists of how much information should be packed into a minimal amount of code. There is a tradeoff between compactness of coding style and readability. A single very long line of code containing function calls and a complicated data access path may be made more understandable by splitting it into several lines and introducing new procedures to accomplish parts of the original line's intention. Because too many procedures and functions declared prior to the main body of a package or subprogram tend to have a cluttered effect and make it harder to follow the thread of what is going on, stubbing these subprograms is also another alternative. These possibilities are further explored in the case study STUBBING AND READABILITY.

On a very detailed level, it should be noted that Ada does provide constructs which enhance the readability of the code. For instance the range syntax eliminates the need to check boundary conditions using compound inequalities. An example of this is given in the case study SUCCINTNESS OF RANGE CONSTRAINT.

Each programmer tends to develop his or her own coding style. For ease of reading the code and maintaining it, some agreement on a standard coding style is needed. Naming conventions should be adopted as well. Dot notation, while it has the advantage of identifying the parent package(s) of a piece of data or a subprogram, does result in more cumbersome code when too many parents are identified. One possibility is to rename all

lengthy parent names at the beginning of subprograms. Another alternative is to develop a tool which will take a program and supply a new version of this program using dot notation. Thus, a programmer could use simpler names, knowing that these names could automatically be expanded to show their source. This tool would also facilitate the maintenance programmer's job because it would enable him to trace back the source package containing a particular subprogram without having the degree of familiarity with the locations of these programs that the implementor had.

5.3 Coding Paradigms

Several of the coding paradigms are related to tasking and rendezvous issues. There are case studies on implementing an exit from a rendezvous, RENDEZVOUS AND EXIT, and on decoupling activities using tasking, DECOUPLING PARTLY INDEPENDENT ACTIVITIES.

A characteristic embedded systems problem deals with memory-mapped I/O. How does the programmer pass the address of an interrupt handler? This issue is addressed in the case study MEMORY-MAPPED I/O IN ADA. Another low-level issue deals with the use of representation specifications. Because the compiler does not guarantee any particular representation, if the system relies on some representation, then the programmer should use the Ada facility for representation specifications.

Good programming style dictates that goto's should be avoided at all possible times. While Ada does provide a goto construct, it also provides many other constructs which could substitute. The case study ELIMINATING GOTO's addresses this issue in greater detail.

5.4 Examples of Ada Constructs

In the course of reviewing the contractor's materials in the technical interchange forums, some interesting uses of Ada constructs came to light. These are presented in the case study ARRAY OF ARRAYS.

An interesting result of the technical interchange forums concerns the use of types, numeric types in particular. Both contractors found that Ada's strong typing facility helped in detecting errors at an early stage. Many of these errors came from missing type conversions among variables of the same underlying numeric types. The excessive number of conversions needed to rectify the problem made a convincing case for the use of subtypes. In general, data which has to be combined in operations frequently should be of the same type, though possibly of different subtypes in order to stress the different ranges that might apply.

5.5 Handling Impossible States

The impossible condition is a well known phenomenon in every software project. The typical example is a case statement in which some of the cases are known to be impossible in a certain context. A different situations arises when the specification in unclear about some of the cases.

This was identified as a typical source or problems in software development when

1. a programmer does not know what to do in the impossible case, so he or she simply ignores the possiblility, or

2. a programmer confuses the logically impossible case with one which was not covered by the specification (which might either be impossible, or might simply reveal an error in the specification).

Technically, at least two approaches exist (and should be taught explicitly to all programmers and designers).

1. Raise an exception (or let the system raise a predefined exception).

2. Call a system wide panic routine (or entry). This is probably what a task would do before terminating, to inform the rest of the system that something highly abnormal has happened.

A serious difficulty was perceived to be that of educating designers and programmers to be aware of the problem. At the beginning of each project the technical mechanisms should be established, but most of all there should be a standard practice whereby a programmer or module designer can promptly call attention to an impossible situation and receive guidance for its handling. As mentioned before, all too often programmers are tempted to hide troublesome cases, rather than seeking a resolution.

5.6 Summary

The code and unit test phase of the life cycle yielded the greatest number of case studies for the report; however, the earlier phases raised many issues for further research.

5.7 <u>Case Studies</u>

The case studies illustrating this phase are:

- Stubbing and readability
- Succintness of range constraint
- Rendezvous and <u>exit</u>
- Decoupling partly independent activities
- Memory mapped I/O in Ada
- Eliminating <u>goto</u>'s
- Array of arrays

STUBBING AND READABILITY

1. BACKGROUND

Case Study Objective

To illustrate the effect of stubbing on code readability.

Designer's Problem

Through stubbing, the system designer is able to postpone the specification of the body of a program unit, thus allowing a textually top-down presentation of the design. What constitutes the correct use of stubbing when designing in Ada?

Discussion

Coding, like detailed design, is an iterative process involving successive refinements to a first draft. While the end product is usually much improved, the programmer runs the risk that elegance and economy are achieved at the expense of readability. Stubbing is often a solution to this problem.

Stubbing promotes readability in the following areas:

a. Structuring (top-down presentation)

b. Introduction of local (to the stub) constants, renaming, etc.

There are negative aspects of stubbing also. These include the following:

a. The creation of a wideband interface is not good for design. The fact that the subunit inherits the visibility of the parent at the place of the stub creates a wide interface with implicit connections.

b. Overuse of stubbing can result in excessive fragmentation of logic.

2. DETAILED EXAMPLE

Example Problem Statement

Within a larger system, there is a process READ_CONTINUOUS which reads message characters from a message.

```
procedure READ+CONTINUOUS(MESSAGE : MSGID;
                          COUNT : in out CHAR+COUNT;
                          WHERE+FROM : in out POSITION;
                          TEXT : out STRING;
                          STATUS : out ERR+STAT) is
----------------------------------------------------------------------
-- NAME: READ+CONTINUOUS
-- PURPOSE: Reads message characters from a message.  The initial
--          position is established by POS, which should be set by
--          OPEN+FOR+READ or some other position-setting procedure.
-- PROGRAMMER:
-- DATE:
-- PARAMETERS:
--              IN:     MESSAGE - which message to read from
--              IN OUT: COUNT - how many characters to read
--                            - on output, how many were actually read
--                      WHERE+FROM - starting position for the read
--                                 - moved forward by the amount of
--                                 - characters read
--              OUT:    TEXT - the characters read
--                      STATUS - OK - no problems encountered
--                               END+OF+TEXT - end of the part
--                               LINKAGE+ERROR - an improperly linked
--                                             - segment was encountered
--                               OTHER+ERROR - any other problem
-- DESCRIPTION: First checks to see if any characters are left to read
--              in the current segment.  If not, advances to the next
--              segment.  If this segment is null, this signals the end
--              of the part.  If this is the last part of the message,
--              END+OF+TEXT is returned.  If not, the routine moves to
--              first segment of the next part.  PART is checked for
--              LINKAGE+ERROR only if the routine has not moved to
--              the next part.  In addition, the routine checks
--              CLASS and ASN to ensure that no LINKAGE+ERROR has
--              occurred.  Once assured of characters to read, the
--              routine determines if it can get all it wants from
--              its current segment.  If it can, it does so and
--              forces an exit from the loop by setting the number
--              of characters left to read to zero.  Otherwise, it
--              reads all the remaining characters in the current
--              segment, updates the number read and remaining to
--              read, and loops back to the start.
-- EXCEPTIONS: All exceptions are handled by returning OTHER+ERROR.
----------------------------------------------------------------------

   NEXT : SEG+PTR;  -- used to go to next segment where required
   REMAINING+TO+READ : CHAR+COUNT;  -- number of characters to read
   THIS+READ : CHAR+COUNT;  -- number of characters to be read this time
```

```
begin -- READ_CONTINUOUS

  REMAINING_TO_READ := COUNT;  -- all characters still to be read
  COUNT := 0;  -- no characters have been read

  while REMAINING_TO_READ > 0 loop  -- main loop

    if WHERE_FROM.COUNT > READ_CHARACTER_COUNT(SEG=>WHERE_FROM.SEG) then
      -- past last char in this segment, advance to the next segment
      NEXT := NEXT_SEGMENT(SEG => WHERE_FROM.SEG);

      if NEXT = null then  -- at the end of the part
        if READ_PART(SEG => WHERE_FROM.SEG) = PART_NAME'LAST then
          -- end of the message
          STATUS := END_OF_TEXT;  -- tell caller that we are done
          exit;
        else  -- advance to the next part
          NEXT := MESSAGE.SECTIONS(PART_NAME'SUCC(READ_PART(SEG =>
              WHERE_FROM.SEG))).FIRST_SEGMENT;
        end if;

      else  -- remaining in the same part, check the linkage

        if READ_PART(SEG=>NEXT) /= READ_PART(SEG=>WHERE_FROM.SEG) then
          STATUS := LINKAGE_ERROR;  -- return error code
          exit;
        end if;

      end if;

      -- check linkage for all new segments
      if READ_CLASS(SEG => NEXT) /= READ_CLASS(SEG => WHERE_FROM.SEG)
          or else
          READ_EASN(SEG => NEXT).ASN /=
          READ_EASN(SEG => WHERE_FROM.SEG).ASN then
        STATUS := LINKAGE_ERROR;  -- return error code
        exit;
      end if;

      -- update position
      WHERE_FROM.SEG := NEXT;
      WHERE_FROM.COUNT := 1;
    end if;  -- done advancing to the next segment

    if REMAINING_TO_READ + WHERE_FROM.COUNT -1 <=
        READ_CHARACTER_COUNT(SEG => WHERE_FROM.SEG) then
      -- able to read all remaining from this segment
      TEXT(COUNT + 1 .. COUNT + REMAINING_TO_READ) :=
          READ_CHARS(SEG => WHERE_FROM.SEG, WHERE => WHERE_FROM.COUNT,
          COUNT => REMAINING_TO_READ);       -- read from segment

      -- update position, # characters read, # remaining to read
      WHERE_FROM.COUNT := WHERE_FROM.COUNT + REMAINING_TO_READ;
      COUNT := COUNT + REMAINING_TO_READ;
      REMAINING_TO_READ := 0;       -- set up for loop exit
    else  -- read all the rest of this segment
      THIS_READ := READ_CHARACTER_COUNT(SEG => WHERE_FROM.SEG) -
          WHERE_FROM.COUNT;   -- calculate number to read
      TEXT(COUNT + 1 .. COUNT + THIS_READ) := READ_CHARS
          (SEG => WHERE_FROM.SEG, WHERE => WHERE_FROM.COUNT,
          COUNT => THIS_READ);  -- read from segment
      -- update position, # characters read, # remaining to read
      WHERE_FROM.COUNT := WHERE_FROM.COUNT + THIS_READ;
      COUNT := COUNT + THIS_READ;
      REMAINING_TO_READ := REMAINING_TO_READ - THIS_READ;
    end if;

  end loop;  -- end of main program loop

exception
  when others =>
    STATUS := OTHER_ERROR;
end READ_CONTINUOUS;
```

FIND_NEXT_SEG

FIRST_SEG_OF_NEXT_PART

From a readability standpoint, the statement which advances the cursor to the first segment of the next part of the message is far too dense:

```
NEXT:=MESSAGE.SECTIONS(PART_NAME'SUCC(READ_PART(SEG=> WHERE_FROM.SEG)))
      .FIRST_SEGMENT;
```

Although not immediately apparent, the source of this assignment is the FIRST_SEGMENT component accessed by an array element in the SECTIONS component of MESSAGE. The index corresponding to this array element is found by applying the successor attribute to the current value of the enumeration type PART_NAME. To find the current value of PART_NAME, the function READ_PART takes WHERE_FROM.SEG as an actual parameter and returns a result of type PART_NAME. PART_NAME is an enumeration type which serves as an index for the array SECTIONS. SECTIONS is part of the record MESSAGE (see Figure 1). Even ignoring the heavy use of access types in this statement and the fact that its declarations must be garnered from three different packages, logically, this code is a maintainer's nightmare.

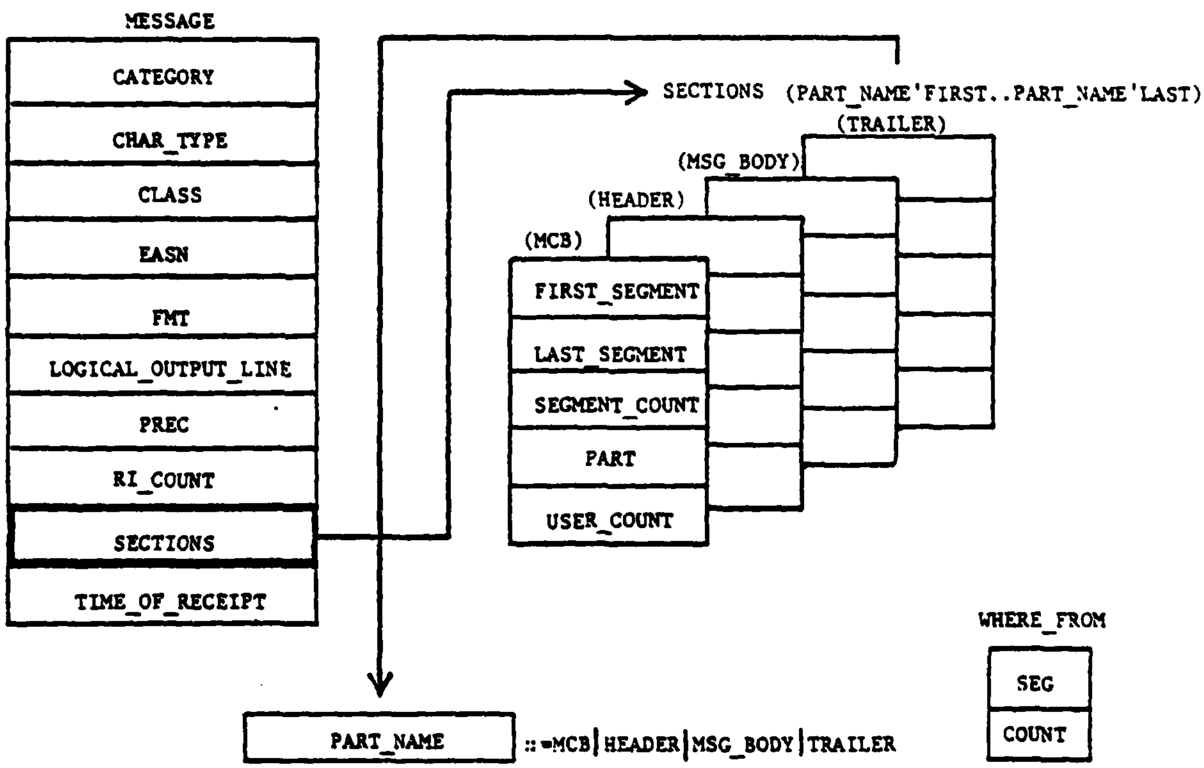


Figure 1. Diagram of Message Data Structure

Solution Outline

To improve readability, the low-level logic of the above statement is best captured in a procedure with local variables and constants. This procedure is then stubbed in the main program.

In the solution below, procedure FIND_NEXT_SEGMENT operates on global variables which are renamed or initialized for clarity. The initial values are computed within expressions which represent subunits of the original logic. FIND_NEXT_SEGMENT first checks for end of segment and end of text conditions and advances to the next segment by means of the local function FIRST_SEG_OF_NEXT_PART if both conditions are false. In addition FIND_NEXT_SEGMENT checks to ensure that no linkage error has occurred. STATUS is a parameter of FIND_NEXT_SEGMENT since the value of STATUS affects the flow of control of the context in which FIND_NEXT_SEGMENT is called. If no parameter is given, the reader of READ_CONTINUOUS will not understand what causes the value of STATUS to change.

Detailed Solution

```
with GLOBAL_TYPES;
with SEGMENT_OPS;
use GLOBAL_TYPES;
use SEGMENT_OPS;

package body MESSAGE_OPS is
     ...

procedure READ_CONTINUOUS(MESSAGE          : MSGID;
                          COUNT            : in out CHAR_COUNT;
                          WHERE_FROM       : in out POSITION;
                          TEXT             : out STRING;
                          STATUS           : out ERR_STAT)      is
     ...
     procedure FIND_NEXT_SEGMENT (STATUS: in out ERR_STAT) is separate;

begin
     REMAINING_TO_READ := COUNT; -- all characters still to be read
     COUNT := 0;                 -- no characters have been read
     STATUS := OK;

     while REMAINING_TO_READ > 0 and STATUS = OK
     loop
          if WHERE_FROM.COUNT>READ_CHARACTER_COUNT(WHERE_FROM.SEG) then
                    FIND_NEXT_SEG (STATUS);
          end if;
               --WHERE_FROM points to non-exhausted segment
          if STATUS = OK and ... then
               ...  -- see existing code
          end if;
     end loop;
```

```
exception
     when others =>
          STATUS := OTHER_ERROR;
end READ_CONTINUOUS;
end MESSAGE_OPS;

separate (READ_CONTINUOUS)
procedure FIND_NEXT_SEGMENT (STATUS: in out ERR_STAT) is
     CURR_SEG: constant SEG_PTR renames WHERE_FROM.SEG;
     NEXT:SEG_PTR := NEXT_SEGMENT(CURR_SEG);
     CURR_PART: constant PART_NAME := READ_PART(CURR_SEG);
     function FIRST_SEG_OF_NEXT_PART return SEG_PTR is separate;

begin
     if NEXT = null then
               --chase the next part if it exists
          if CURR_PART = PART_NAME'LAST then
               STATUS := END_OF_TEXT;
          else
               NEXT := FIRST_SEG_OF_NEXT_PART;
          end if;
     else
          if READ_PART(NEXT) /= CURR_PART then
               STATUS := LINKAGE_ERROR;
          end if;
     end if;
     -- check linkage for all new segments
     if STATUS = OK and ... then
          STATUS := LINKAGE_ERROR;
     end if;
     ...
end FIND_NEXT_SEGMENT;

separate (READ_CONTINUOUS.FIND_NEXT_SEGMENT)
function FIRST_SEG_OF_NEXT_PART return SEG_PTR is
     NEXT_PART: constant PART_NAME := PART_NAME'SUCC(CURR_PART);

begin
     return MESSAGE.SECTIONS(NEXT_PART).FIRST_SEGMENT;
end FIRST_SEG_OF_NEXT_PART;
```

3. EPILOGUE

As the above example illustrates, stubbing is a straightforward way to improve the readability of complex code or detailed design. It should be kept in mind however, that the transformation in Ada from the original logic to a procedure or function is not always a mechanical process. Exit and accept statements in the original module cannot merely be placed in the stub. For example,

```
procedure P is
begin
     OUTER:
     loop
          S1;
          INNER:
          loop
               S2;
               exit OUTER when C1;
               S3;
               S4;
          end loop INNER;
     end loop OUTER;
end P;
```

In order to stub the sequence of code in the inner loop containing S2 and S3, the exit statement must be transformed. The stubbed procedure must set a flag which in time can be tested in an exit condition:

```
OUTER:
loop
     S1;
     INNER:
     loop
          stub (exit_flag);
          exit OUTER when exit_flag;
          S4;
     end loop INNER;
end loop OUTER;

separate (P)
procedure stub (exit_flag : out BOOLEAN) is
begin
     S2;
     exit_flag := not C1;
     if C1 then
          S3;
     end if;
end stub;
```

Trying to stub reveals the need for packages. For example, the stubs in the exercise should be written in a more general, reusable style. This, in turn, points out that a further level of abstraction would be desirable.

THIS PAGE INTENTIONALLY LEFT BLANK

SUCCINTNESS OF RANGE CONSTRAINT

1. BACKGROUND

Case Study Objective

To illustrate how the Ada range syntax expresses a range with greater conciseness and readability than a series of inequalities.

Designer's Problem

The boundary conditions for a value must be tested. These conditions can be checked using inequalities, but is this the best way to code the Ada solution?

Discussion

Testing boundary conditions may be done in two ways in Ada: using the range syntax and using inequalities. In many older programming languages, a combination of inequalities had to be written in order to test boundary conditions. This question to some extent reflects the designer's experience with such languages, insofar as the first solution that presented itself used inequalities.

2. DETAILED EXAMPLE

Example Problem Statement

Boundary checks are made on the altitude value in a radar report to determine the altitude gate. The original code contained the following statement:

```
if R.ALT <TR.ALT + TR.ALTGATE and R.ALT > TR.ALT - TR.ALTGATE then
     if TR.ALTGATE  > ALTGATE'FIRST then
          TR.ALTGATE := TR.ALTGATE - 500;
     end if;
else
     ...
end if;
```

Solution Outline

The condition in the *if* statement above is rewritten using the Ada range syntax. The problem is of the form:

$$x < a + b \quad \text{and} \quad x > a - b$$

Upon closer examination, this condition is equivalent to:

$$a - b < x < a + b$$

From the nested *if* statement above, the reader can observe that the underlying numeric base type of the .ALT and .ALTGATE components of the radar report record R is INTEGER. Therefore, the strict inequality may be rewritten as:

```
a - b + 1  <=  x  <=  a + b - 1
```

This last form is the mathematical equivalent of the Ada expression

```
x  in  a - b + 1  ..  a + b - 1
```

and is shown in the detailed solution.

Detailed Solution

```
if R.ALT in TR.ALT - TR.ALTGATE + 1 .. TR.ALT + TR.ALTGATE - 1 then
     if TR.ALTGATE > ALTGATE'FIRST then
          TR.ALTGATE := TR.ALTGATE - 500;
     end if;
else
     ...
end if;
```

3. EPILOGUE

The Ada range syntax provides a succinct way of expressing a boundary check. The inequality

```
a - b + 1  <=  x  <=  a + b - 1
```

may be rewritten as

```
-b + 1  <=  x - a  <=  b - 1
```

It remains as an exercise for the reader to write the corresponding Ada solution.

RENDEZVOUS AND EXIT

1. BACKGROUND

Case Study Objective

To illustrate how the programmer achieves the effect of exiting from a rendezvous, given that the language forbids an actual exit from inside a rendezvous.

Designer's Problem

Under certain conditions, the designer must terminate a rendezvous early and exit the loop containing this rendezvous. Because the language does not allow a direct exit from the rendezvous, the question arises, can this effect nevertheless be achieved in Ada?

Discussion

The problem here involves finding a means to transfer the flow of control efficiently and smoothly to an appropriate point.

2. DETAILED EXAMPLE

Example Problem Statement

A task that outputs messages consists of several states: a ready state, a validation state, and a send state. In the ready state, the process seeks the next message to be sent. The validation state performs message validation and the message is transmitted during the send state. At any time, the message currently being processed may be preempted. In particular, for the send state, the PDL follows:

```
SEND:
while there are messages to send
loop
     give message to transmitter task;
     loop
          if preempted with higher priority message then
               if header of new message is valid then
                    save original message;
                    preempt task sending original message;
               end if;
          elsif preempted without higher priority message then
               save original message;
               preempt task sending original message;
          end if;
     end loop;
-- exit when no more messages to send, i.e., all preempting and
                                 -- original messages have been sent
end loop;
```

The basic function of the task being studied here is to buffer preemption requests from the task that is managing the queues of messages to be sent and the task that is actually transmitting the message (see the case study TASK PREEMPTION). Figure 1 is a structure chart showing the context of the preemption buffering task. However, this task not only buffers preemption requests; it also buffers a single message to be sent on a particular physical line.

In the implementation, PREEMPT_MESSAGE is called by the queueing task to indicate that the current message being sent is to be preempted. If the NEW_MSG argument of PREEMPT_MESSAGE is non-null, a higher priority message is to be sent instead. FINISHED_SENDING is called when the sending task has completed sending a message over a physical line.

A problem occurs when a message is preempted by a null message and the transmitter notifies the sender that the (original) message is completely transmitted through a call to FINISHED_SENDING. There is a nested rendezvous structure here which must be exited:

```
READY_STATE:
loop
     ...
     SEND_STATE:
     loop
          select
               accept PREEMPT_MESSAGE ( ...) do
                    if NEW_MSG = null then
                         select
                              accept PREEMPT_CHECK ( ... );
                         or
                              accept FINISHED_SENDING;
                         end select;
                    else
                         ...
                    end if;
               end PREEMPT_MESSAGE;
          end select;
          ...
     end loop SEND_STATE;
end loop READY_STATE;
```

Because the FINISHED_SENDING call is accepted during the rendezvous with PREEMPT_MESSAGE, the flow of control cannot jump out of the loop enclosing this rendezvous. The design requires that the flow of control after accepting FINISHED_SENDING returns to the READY_STATE. How can this requirement be met in Ada?

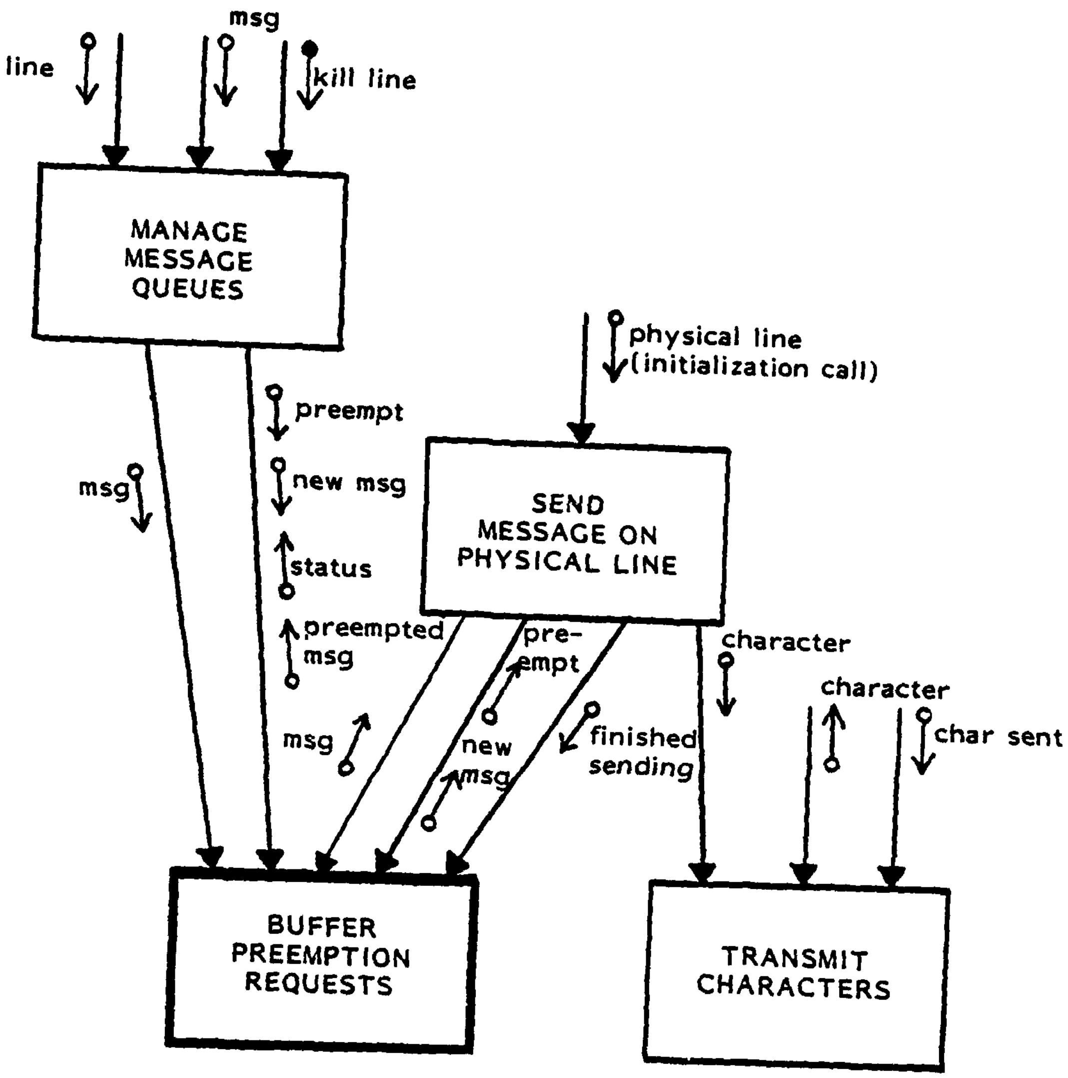


Figure 1. Structure Chart Showing Context of Task for Buffering Preemption Requests.

Solution Outline

The solution to this problem is to introduce an exit flag. The flag is set when the original message is completely sent or when a null message preempt has occurred:

```
READY_STATE:
loop
     ...
     MESSAGES_TO_SEND := TRUE;
     SEND_STATE:
     while MESSAGES_TO_SEND
     loop
          select
               accept PREEMPT_MESSAGE ( ... ) do
                    MESSAGES_TO_SEND := NEW_MESSAGE = null;
                    if NEW_MSG = null then
                         select
                              accept PREEMPT_CHECK ( ... );
                         or
                              accept FINISHED_SENDING;
                         end select;
                    else -- higher priority message to send
                         ...
                    end if;
               end PREEMPT_MESSAGE;
          or
               accept FINISHED_SENDING;
               MESSAGES_TO_SEND := FALSE;
          end select;
     end loop SEND_STATE;
end loop READY_STATE;
```

Because the entire rendezvous consists of the single <u>if</u> ... <u>then</u> ... <u>else</u> statement, once the exit flag here is set, control is immediately transferred to the end of the rendezvous, and then out of the rendezvous. The full solution follows in the next section.

Detailed Solution

```
task body BUFFER_PREEMPTION_REQUESTS IS
-----------------------------------------------------------------------

-- NAME:  BUFFER_PREEMPTION_REQUESTS
-- PURPOSE:  acts as coordinator between the translated message queue
             and the task.  Sending characters to a destination (the send
             task)
   begin  -- OUTPUT_MESSAGE
     -- find out what physical line this is
     accept INIT(LINE : PHYSICAL_LINE) do
       PHYS_LINE := LINE;
     end;

     loop   -- main loop
   <<READY_STATE>>
       -- not currently sending a message
       -- need to get next message
       loop  -- ready state loop
          . . .
       end loop;

   <<VALIDATION_STATE>>
       loop
          . . .
       end loop;

   <<SEND_STATE>>
       -- we have a valid message to send
       accept GET_MESSAGE(MSG : out MSGID) do
         MSG := MESSAGE; -- give the message to the send task
       end;
       EXIT_FLAG := FALSE;
       loop  -- we stay in this loop as long as we have a message to send
         -- During sending, two things can occur:
         --   1--We may be preempted,
         --   2--Send may finish sending the message
         --      (this category includes rejection by the receiver).
         select
           -- check for preemption
           accept PREEMPT_MESSAGE(NEW_MSG : MSGID;
                                  OLD_MSG : out MSGID;
                                  STATUS : MSG_STAT) do
             if NEW_MSG = null then
               -- on null preempts, do not validate
               select
                 -- The option here is in case send finishes the message
                 --   before we finish validation.
                 accept PREEMPT_CHECK(MSG : out MSGID) do -- preempt send
                   MSG := null;  -- tell send it's a null message
                   OLD_MSG := MESSAGE;  -- geve back the old message
                   STATUS := NORMAL_ACC; -- report what happened
                 end;
               or
```

```
            accept FINISHED_SENDING;  -- send completed the message
            OLD_MSG := null; -- no old message
            STATUS := COMPLETED_ACC;  -- tell queue what happened
            EXIT_FLAG := TRUE;  -- go directly to the ready state
                                -- (the flag triggers a loop exit)
                                -- there is no need to even tell send
                                -- about the null preempt, since
                                -- he has no action
          end select;
          MESSAGE := null;
        else -- NEW_MSG /= null
          if HEADER_VALID(NEW_MSG) then  -- validate the new message
            select
              -- The same two situations can occur here as above
              accept PREEMPT_CHECK(MSG : out MSGID) do -- preempt send
                MSG := NEW_MSG;  -- get new message
                OLD_MSG := MESSAGE;  -- pass back old message
                STATUS := NORMAL_ACC;  -- report status to queue
              end;
            or
              accept FINISHED_SENDING;  -- send has finished
                                        --   the old message,
                                        --   give him the validated
                                        --   preempt as a new message
              accept GET_MESSAGE(MSG : out MSGID) do
                MSG : NEW_MSG;  -- give send the new message
                OLD_MSG := null; -- there is no old message
                STATUS := COMPLETED_ACC; -- tell queue what happened
              end;
            end select;
            MESSAGE := NEW_MSG;  -- save the msgid of the new message
                                 -- for possible preemption
                                 -- from here, flow of control goes
                                 -- around the send loop to the top
          else -- header is not valid, do not preempt send
            OLD_MSG := null; -- no old message
            STATUS := REJ;   -- tell queue we rejected his message
          end if;
        end if;
      end;  -- accept PREEMPT_MESSAGE
      exit when EXIT_FLAG;
    or
      -- finished sending the message without preemption
      accept FINISHED_SENDING;
      exit; -- exit the send loop, go back to ready state
    end select;
  end loop;  -- while sending
end loop;  -- main loop

end OUTPUT_MESSAGE;
```

3. EPILOGUE

Implementing some coding paradigms in Ada requires additional control structures or variables which are not visible at the design or the PDL level. The effect of exiting from a rendezvous is such an example.

THIS PAGE INTENTIONALLY LEFT BLANK

DECOUPLING PARTLY INDEPENDENT ACTIVITIES

1. BACKGROUND

Case Study Objective

To show how tasks can be used to implement the decoupling of two activities in Ada.

Designer's Problem

A log of each activity executed in a subsystem must be kept, yet this logging function should not interfere with or delay the rest of the processing. How is the requirement best expressed in Ada?

Discussion

The logging function is essentially an independent, concurrent process. This function must interface once with the activity to be logged in order to receive necessary statistics, but outside of this single interface, it can operate independently of other processes. The solution presented here will show how Ada tasks lend themselves to this application.

2. DETAILED EXAMPLE

Example Problem Statement

For each message received, processed, or transmitted in a subsystem, a log must be made that records the number of blocks sent, the line on which it was sent, the message itself, and a sequence number. Independent and concurrent activities receive, process, or transmit messages, and these operations operate under stringent timing constraints.

Solution Outline

At first glance, one could just call a logging procedure after a message is complete. For a transmit operation, the PDL might look like:

```
loop
     transmit message;
     log(blocks, line, message, sequence number);
     read next message;
end loop;
```

This solution, however, delays the main transmission process inasmuch as a new message can be neither read nor transmitted until the previous message is fully logged. The independence and potential concurrency of the logging procedure is not exploited.

Alternatively, the logging procedure could be represented as a task. The message information is passed during a single rendezvous, after which the logging is continued concurrently and independently of the other message processing. While this solution does allow for the logging to be concurrent with the other message operations, it is not problem free. Depending on the frequency of logging requests and the speed with which the logging task can fulfill them, the various message operations will of necessity be delayed until the logging task is ready to accept the data.

A third solution is to create a new task for each logging request. Thus, no message operation needs to wait until a logger is ready to accept its information. A task terminates if it has no statements left to execute and it has no dependent tasks. All of these logging tasks will therefore die when they complete their function. Because many tasks, as opposed to a single task, are needed, a logging task type is defined. Furthermore, because these tasks should only be activated on an as needed basis, an access type to these tasks is defined. A logging task is therefore activated through the execution of an allocator. After the rendezvous, in which the logging task receives the message, the pointer to the task is deallocated because no further interaction with this task will be needed. Another benefit of deallocating the pointer is that it will be available to point to the next logging task. Only the detailed solution for the transmit operation is presented.

Detailed Solution

```
with DEFS; use DEFS;
package PHYSICAL_PORT is

     type LOG_REQUEST is
          record
               BLOCKS     : NUM_BLOCKS;
               LINE       : PHYSICAL_LINE;
               MSG        : MSG_ID;
               SEQ_NUM    : CSN;
          end record;

     task TRANS is
          entry MIT(MESSAGE : MSG_ID;
                    LINE    : PHYSICAL_LINE);
     end TRANS;

     task type SEND_MSG is
          entry INIT(LINE : PHYSICAL_LINE);
          entry GET_MESSAGE(MSG : in MSG_ID);
     end SEND_MSG;
```

```
     task type LOG_FUNCTION is
          entry LOG(REQ : LOG_REQUEST);
     end LOG_FUNCTION;

     type LOG_PTR is access LOG_FUNCTION;

end PHYSICAL_PORT;

package body PHYSICAL_PORT is

task body SEND_MSG is

     LOG_ACCESS      : LOG_PTR;
     MESSAGE         : MSG_ID;
     LINE_OUT        : PHYSICAL_LINE;

     procedure READ_BLOCKS is separate;
     procedure READ_SEQ_NUM is separate;

begin
     accept INIT(LINE : PHYSICAL_LINE) do
          LINE_OUT := LINE;
     end INIT;
     loop
          accept GET_MESSAGE(MSG: MSG_ID) do
               MESSAGE := MSG;
          end GET_MESSAGE;
          READ_BLOCKS;
          READ_SEQ_NUM;
          TRANS.MIT(MESSAGE,LINE);
          LOG_ACCESS := new LOG_FUNCTION;
          LOG_ACCESS.LOG((BLOCKS,        -- double parentheses
                          LINE_OUT,      -- needed because you
                          MESSAGE,       -- are passing a single
                          SEQ_NUM));     -- parameter
          LOG_ACCESS := null;
     end loop;
end SEND_MSG;

task body LOG_FUNCTION is
          ...
end LOG_FUNCTION;

task body TRANS is
          ...
end TRANS;

begin
...
end PHYSICAL_PORT;
```

3. EPILOGUE

Creating tasks "on the fly" eliminates all unnecessary dependence between the logging function and the message operations. By limiting the dependence to the transfer of message information, the potential for concurrent and independent execution is best exploited. However, creating and terminating tasks may be expensive in some Ada implementations.

An alternative approach is for the LOG_FUNCTION to maintain an internal queue of information to be logged. The LOG entry would then queue the items to be logged, and a separate task would actually do the logging by removing entries from the queue. This alternative approach should be tried as an exercise.

With the alternate approach, the code written to log information would consist just of the LOG_FUNCTION.LOG entry call. How can the current code be changed so the choice between those implementation approaches can be deferred until it is known which is sufficiently efficient? (In essence, logging should be performed by a procedure call; the body of the procedure determines the implementation approach. If the procedure is called in-line, the approach will be no less efficient than the current approach. This shows how to retain the ability to choose between alternative implementation approaches.)

MEMORY-MAPPED I/O IN ADA

1. BACKGROUND

Case Study Objective

The primary objective of this case study is to recommend an approach to doing low-level, memory-mapped I/O in Ada. Secondary objectives are to demonstrate the use of address representation specifications, shared variables, and the standard package LOW_LEVEL_IO.

Designer's Problem

In implementing embedded computer systems it will often be necessary to program low-level device I/O in Ada. The designer's problem discussed here is to select the Ada features best suited to handle low-level I/O on a machine using memory-mapped I/O (see discussion below for an explanation of memory-mapped I/O).

Discussion

Embedded computer systems typically consist of a number of special peripheral devices under control of a computer. In fact, the reason for the existence of the computer system is often to provide flexible control of the various peripheral devices. Because of the central importance of specific device control, it is essential that Ada support low-level control of data transfer to and from special I/O devices.

In a memory-mapped I/O computer system, I/O is done by directing ordinary memory reference instructions to a special memory address (such an address is often referred to as a memory-mapped I/O port). A memory read (load) instruction referencing the device-specific memory address causes an input operation from the device. The memory read instruction returns the value input from the device. Except for possible timing differences, the memory read instruction has the same effect as if the referenced location was an ordinary memory location containing the value input from the device.

Conversely, a memory write (store) instruction directed to the special memory address causes a device output operation. The data that the instruction would ordinarily store into the memory location is instead output to the device associated with the referenced location.

Note that the value read from a memory-mapped I/O port is under control of the I/O device. In particular, there is no guarantee that a value stored into such a port address will be returned by a subsequent load instruction.

2. DETAILED EXAMPLE

Example Problem Statement

To illustrate clearly the principles of low-level I/O, the example problem chosen is a particularly simple one: To write an 8-bit value to a memory-mapped I/O device and subsequently to read an 8-bit input value from the device.

Solution Outline

Three alternative approaches to the problem are presented. The first alternative is an obvious, but incorrect, attempted solution. The second alternative is a correction of the first approach. Finally, the third approach presents the recommended low-level, memory-mapped I/O approach. Following the presentation of each alternative is a discussion of the strengths and weaknesses of the approach.

The first and second approaches are based on the use of address representation specifications while the third and recommended approach uses the standard package LOW_LEVEL_IO. The second approach illustrates a use of the standard generic procedure SHARED_VARIABLE_SYNCHRONIZE.

Detailed Solution

Each of the alternative solutions assume the following main program:

```
procedure MAIN is
     subtype BYTE is INTEGER range 0..255;
     VAL1, VAL2 : BYTE;

     procedure DEVICE_IO (OUTVAL : in BYTE;
                          INVAL :out BYTE)
     is separate;

begin

VAL1 := 41;                    -- random example value
DEVICE_IO (VAL1, VAL2);
end MAIN;
```

The separate procedure DEVICE_IO is to be implemented. Each of the three approaches to the implementation of DEVICE_IO is described below.

First Approach

The first approach to implementing DEVICE_IO is to define a variable whose address is specifically designated to be that of the memory-mapped IO_PORT. Output to the device is then done by assignment to the variable and input from the device is done by the use of the variable in an expression. The fatal flaw in this straightforward approach is discussed after the listing of the attempted solution.

```
-- first approach (INCORRECT)

separate (MAIN)
procedure DEVICE_IO (OUTVAL : in BYTE;
                     INVAL  : out BYTE) is

    IO_PORT     : BYTE;                    -- The memory_mapped I/O port
    IO_PORT_LOC : constant := 16#C0F0#;   -- The address of the port

    for IO_PORT use at IO_PORT_LOC;  -- an address representation
                                     -- specification telling the
                                     -- compiler that the variable
                                     -- io_port is to be located at
                                     -- address IO_PORT_LOC

begin

    IO_PORT := OUTVAL;     -- write outval to the io_port
    INVAL   := IO_PORT;    -- read inval from the io_port

end DEVICE_IO;
```

The difficulty with this approach lies in the fact that IO_PORT does not behave like an ordinary variable, since the value stored into the variable is not necessarily the value that will be subsequently read back. The compiler, not knowing this, will note that IO_PORT is local to the procedure and so the two assignments are logically equivalent to

```
INVAL := OUTVAL;
```

There is no need to assign to the local variable at all, and so there is no need to read the local variable either. Equally well, the compiler might keep OUTVAL in a register and merely store this register value into the variable INVAL. There will be no explicit read of the value in IO_PORT and therefore no input operation from the device.

Second Approach

To correct the problem with the first approach, note that the variable IO_PORT behaves as if it were shared between two tasks -- the task running the main program and an imaginary task carried out by the I/O device. Since the imaginary I/O task can store a value into the variable IO_PORT, the main task cannot consider IO_PORT to be a variable accessible only by the procedure. In particular, it cannot depend on the value of the variable being the last value it stored into the variable.

To handle this kind of situation, Ada includes a built-in generic procedure, SHARED_VARIABLE_SYNCHRONIZE. This procedure ensures that a variable is explicitly updated with its latest value (rather than merely holding that value in a register). It also ensures any subsequent reference to the variable will cause an explicit read of the value of the variable.

The use of (a suitable instantiation of) the SHARED_VARIABLE_SYNCHRONIZE procedure after an assignment to IO_PORT will ensure that an explicit store is done. Likewise, a call to the procedure before a use of IO_PORT in an expression will ensure an explicit read from the variable. In the simple example at hand, the procedure call after the store can be combined with the call before the read so that only a single call is required.

```
-- second approach
with SHARED_VARIABLE_SYNCHRONIZE;
separate (MAIN)
procedure DEVICE_IO (OUTVAL : in BYTE;
                     INVAL  : out BYTE) is
     IO_PORT     : BYTE;
     IO_PORT_LOC : constant := 16#C0F0#;

     for IO_PORT use at IO_PORT_LOC;

     procedure EXPLICIT_IO is new SHARED_VARIABLE_SYNCHRONIZE (BYTE);

begin
     IO_PORT := OUTVAL;        -- write to IO_PORT
     EXPLICIT_IO (IO_PORT);    -- ensure explicit write

     EXPLICIT_IO (IO_PORT);    -- ensure subsequent explicit read (could
                               -- be collapsed with preceding call)
     INVAL := IO_PORT;         -- read from IO_PORT

end DEVICE_IO;
```

This alternative solution should work; however, it suffers from a certain deviousness. Things are not what they seem to be -- an assignment statement, or even an innocent expression, is really an I/O operation. This solution is probably acceptable if accompanied by appropriate comments and if restricted to subprograms specifically dealing with device I/O operations. There is, however, a better solution presented in the next section that explicitly indicates what is happening.

Third Approach

The third, and recommended, approach uses the standard package LOW_LEVEL_IO. This implementation dependent package is specifically intended for use in detailed control of peripheral devices. The implementor of an Ada system targeted to a memory-mapped computer system should include suitable device types and procedures for handling the memory-mapped devices.

The specification of LOW_LEVEL_IO that is assumed for this example is shown below.

```
with SYSTEM;
package LOW_LEVEL_IO is
     type DEVICE_CLASS is (PORT_DEVICE, MEMORY_DEVICE);
     subtype PORT_ADDRESS_TYPE is SYSTEM.address range 0..255;
     subtype MEMORY_ADDRESS_TYPE is SYSTEM.address range 16#C0C0#..16#C0FF#;

     subtype DATA_TYPE is INTEGER range 0..255;

     type DEVICE_TYPE (CLASS : DEVICE_CLASS) is
          record
               case CLASS is
                    when PORT_DEVICE   =>
                         PORT_ADDRESS   : PORT_ADDRESS_TYPE;
                    when MEMORY_DEVICE =>
                         MEMORY_ADDRESS : MEMORY_ADDRESS_TYPE;
               end case;
          end record;

     procedure SEND_CONTROL    (DEVICE  : DEVICE_TYPE;
                                DATA    : IN DATA_TYPE);
     procedure RECEIVE_CONTROL (DEVICE  : DEVICE_TYPE;
                                DATA    : OUT DATA_TYPE);

end LOW_LEVEL_IO;
```

The revision of device_IO using the LOW_LEVEL_IO package is shown below.

```
-- third, recommended approach
with LOW_LEVEL_IO;
separate (MAIN)
procedure DEVICE_IO (OUTVAL : in BYTE;
                     INVAL  : out BYTE) is

     IO_PORT_LOC : constant LOW_LEVEL_IO.MEMORY_ADDRESS_TYPE
                        := 16#C0F0#;
     IO_PORT     : constant LOW_LEVEL_IO.DEVICE_TYPE
                        := (LOW_LEVEL_IO.MEMORY_DEVICE, IO_PORT_LOC);

begin
     LOW_LEVEL_IO.SEND_CONTROL (IO_PORT, OUTVAL);    -- write outval

     LOW_LEVEL_IO.RECEIVE_CONTROL (IO_PORT, INVAL);  -- read inval

end DEVICE_IO;
```

This solution is recommended because it explicitly shows that I/O operations are being performed. The declarations of the two constants allow the actual I/O operations to be written cleanly and clearly.

3. EPILOGUE

This simple example has illustrated two acceptable ways of doing low-level device control for memory-mapped I/O devices (and one possible pitfall). It should make clear even to assembly language programmers that low-level I/O routines can be written in Ada. As further illustration of the point, an exercise could be defined to use low-level I/O to implement a complete device driver for a simple device (such as a UART). Other points that should be covered include techniques for fielding interrupts from I/O devices.

ELIMINATING GOTOs

1. BACKGROUND

Case Study Objectives

This case study uses an example drawn from actual code to show how the elimination of goto statements improves the clarity of the code. It illustrates a standard technique from removing gotos. It also illustrates how use of PDL (program design language) can improve understandability, and how detailed statements in the PDL version may be realized in code quite differently from the ways suggested by the PDL. This illustrates why PDL will differ from actual code.

Designer's Problem

The code used here is that part of a message switching system responsible for sending a particular message's characters over a particular physical line attached to the switch. The program checks every so often to see if there is a higher priority message to send, in which case, it cancels the transmission of the current message, and begins transmitting the higher priority message.

2. DETAILED EXAMPLE

Example Problem Statement

PDL for the original code is given in Figure 1. We created this PDL after studying the original code. (The contractor did not produce PDL descriptions of system modules.) The use of labels in the PDL conforms to the use of labels in the original code (see Figure 5). In essence, the labels describe various states encountered in the process of transmitting a message. There are two goto statements in this code, and the problem is how to eliminate them.

Detailed Solution

The second goto statement is easily eliminated. When a goto statement occurs at the end of the then alternative in a conditional statement, the first step in eliminating the goto is to convert the following statements into an else alternative. In this case, it happens that this is sufficient to eliminate this goto statement.

Figure 2 shows the rewritten version. Notice that we have replaced the goto statement with an else alternative and a comment indicating the expected flow of control.

The remaining goto statement exits from a loop, so if this statement is to be removed, we must specify the conditions under which this loop is to be executed. In particular, we want to send a character from the message as long as the transmission has not been preempted, and as long as there are characters to send. Since Ada does not permit us to combine the for and while loop forms, we must replace the for loop of Figure 2 with:

```
while not preempted AND have characters to send
```

We have capitalized AND to indicate that we are using Ada's logical operator rather than natural language. If upper and lower case were not available, it would be reasonable to parenthesize the operands:

```
WHILE (NOT PREEMPTED) AND (HAVE CHARACTERS TO SEND)
```

to indicate this distinction.

Note that our use of PDL implies that the "check for preemption" operation sets some kind of indicator saying that the transmission is preempted (see Figure 3). Moreover, this indicator is implicitly set to "not preempted" before the loop is entered the first time. We have chosen not to clutter the PDL with explicit statements setting and clearing a "preempted" variable. This might be criticized during a design review, depending on how clear the implicit need to reset the flag was.

If it were considered more understandable to add the statements, then we would add:

```
preempted := false;
```

after "transmit message id", and we would (see Figure 4) replace the "check for preemption" call with:

```
check for preemption (preempted);   -- sets preempted
```

Having now expressed the conditions for executing the loop that sends individual characters, we must now consider the fact that the goto statement introduces a loop into the program. Since a loop is introduced and we want to eliminate the goto statement, we must insert a loop statement at the appropriate point, namely, after the label "start of message sequence." We must also state under what conditions this loop is to be executed. Initially, it is executed because we have a message to transmit, but it might also be executed because message transmission was preempted. So we simply state these two conditions:

```
while have message to transmit OR preempted
```

As we will see, this does not mean that we have to have two Boolean variables in our executable code, but since there are two reasons for executing the loop, it is clearest to state this fact as straight-forwardly as possible.

Should explicit assignments be made to "have message to transmit?" It is probably more in the spirit of PDL not to make explicit assignments in this case; the reader will know from the logic of the program whether there is a message to transmit or not. The answer may also depend on whether the person writing the PDL will also be the person writing the code.

The variables "have message to transmit" and "preempted" are state variables. Such variables typically need to be introduced when goto statements are removed from programs. They capture the conditions under which certain actions are to be performed. Reading Figure 3, it should be much more apparent how preemption affects the flow of control, and some might argue that Figure 4 makes it even more clear. If you prefer Figure 4 (with its explicit assignments to state variables) to Figure 3, then consider Figure 3 an intermediate step in arriving at Figure 4. One would write Figure 3 first and then, after being satisfied with the overall logic, insert the appropriate state variable assignments.

Notice also, how we have replaced all the labels in the original PDL. Sometimes we have used comments expressing the various states of message transmission, and sometimes we have used loop names, so that the names are checked at the end of the loop.

The original code for this task is given in Figure 5, although we have changed some of the statement labels. The code produced for our goto-free PDL is given in Figure 6. In this code, we have assumed that the STATUS variable includes a value called "preempted". Notice how the STATUS variable serves the function of the "have message to send" variable as well as the "preempted" variable in the PDL. This shows how the flow of control indicated in the PDL need not be slavishly followed when writing the actual code. Note also that the process of checking for preemption produces the message to be sent in place of the message currently being transmitted. Hence, the action of getting the higher priority message (see Figures 3 or 4) is not actually needed. Even though we knew that checking for preemption produced the new message to send, we separated out these actions in the PDL because this seemed to make the logic clearer. Combining these actions is a way of making the implementation more efficient, and is typical of the kinds of changes one would expect to see when translating PDL into actual code. Also note how we used an exit statement to ensure exit when either the message has been preempted or we were unable to transmit the complete message (STATUS /= OK).

3. EPILOGUE

This exercise presented ways of eliminating goto's which in effect improved the clarity of the code. An interesting lesson here is the usefulness of writing PDL in developing actual code.

```
task body send_mode_II_and_IV is
begin
<<startup>>
     notify operator of physical line startup;

     loop -- forever
<<get next message>>
          get message to transmit;

-- transmit message
<<send start of message sequence>>
          transmit message id;
          log start of message transmission;

<<send message text>>
          for all characters in message
          loop
               send character;
               check for preemption;
               if preempted then
                    -- transmit higher priority message
                    send cancel-transmission sequence;
                    log cancellation;
                    get higher priority message to send;
                    GOTO SEND START OF MESSAGE SEQUENCE;
               end if;
          end loop; -- all characters in message

<<check for errors>>
          if unable to find complete message then
               send cancel-transmission sequence;
               log cancellation;
               send suspended transmission message;
               notify operator;
               -- give up on this message; get next message
               GOTO GET NEXT MESSAGE;
          end if;

<<send end of message sequence>>
          send end of message sequence;
          log end of message transmission;
          increment count of successful message transmissions;

<<send trailer sequence>>
          if not sending interswitch message then
               send trailer sequence;
          end if;
          set table showing line is available;
          -- ready to transmit next message
     end loop;  -- forever

end send_mode_II_and_IV;
```

Figure 1. PDL for Original Code

```
task body send_mode_II_and_IV is
begin
<<startup>>
     notify operator of physical line startup;

     loop -- forever
<<get next message>>
          get message to transmit;

-- transmit message
<<send start of message sequence>>
          transmit message id;
          log start of message transmission;

<<send message text>>
          for all characters in message
          loop
               send character;
               check for preemption;
               if preempted then
                    -- transmit higher priority message
                    send cancel-transmission sequence;
                    log cancellation;
                    get higher priority message to send;
                    GOTO SEND START OF MESSAGE SEQUENCE;
               end if;
          end loop; -- all characters in message

<<check for errors>>
          if unable to find complete message then
               send cancel-transmission sequence;
               log cancellation;
               send suspended transmission message;
               notify operator;
               -- give up on this message; get next message
          else

<<send end of message sequence>>
               send end of message sequence;
               log end of message transmission;
               increment count of successful message transmissions;

<<send trailer sequence>>
               if not sending interswitch message then
                    send trailer sequence;
               end if;
               set table showing line is available;
               -- ready to transmit next message
          end if;
     end loop;  -- forever

end send_mode_II_and_IV;
```

Figure 2. Rewritten PDL Eliminating One GOTO

```
task body send_mode_II_and_IV is
begin
     notify operator of physical line startup;

get next message:
     loop -- forever
          get message to transmit;
          have message to transmit := true;
          preempted := false;

send message:
          while have message to transmit OR preempted
          loop
               transmit message id;
               preempted := false;     -- needed when preempted is true

send message text:
               while not preempted AND have characters to send
               loop
                    send next character;
                    check for preemption (preempted);  -- sets preempted
               end loop send message text;

               if preempted then
                    send cancel-transmission sequence;
                    log cancellation;
                    get higher priority message to send;
               else
                    have message to transmit := false;
               end if;
          end loop send message;

-- check for errors
          if unable to send complete message then
               send cancel-transmission sequence;
               send suspended transmission message;
               notify operator;
               -- give up on this message; get next message
          else

-- send end of message sequence
               send end of message sequence;
               log end of message transmission;
               increment count of successful message transmissions;

-- send trailer sequence
               if not sending interswitch message then
                    send trailer sequence;
               end if;
               set table showing line is available;
               -- ready to transmit next message
          end if;
     end loop get next message;  -- forever

end send_mode_II_and_IV;
```

Figure 3. GOTO-Free PDL

```
task body send_mode_II_and_IV is
begin
     notify operator of physical line startup;

get next message:
     loop -- forever
          get messge to transmit;
          have message to transmit := true;
          preempted := false;

send message:
          while have message to transmit OR preempted
          loop
               transmit message id;

send message text:
               while not preempted AND have characters to send
               loop
                    send next character;
                    check for preemption (preempted);  -- sets preempted
               end loop send message text;

               if preempted then
                    get higher priority message to send;
               else
                    have message to transmit := false;
               end if;
          end loop send message;

-- check for errors
          if unable to send complete message then
               send cancel-transmission sequence;
               send suspended transmission message;
               notify operator;
               -- give up on this message; get next message
          else

-- send end of message sequence
               send end of message sequence;
               increment count of successful message transmissions;

-- send trailer sequence
               if not sending interswitch message then
                    send trailer sequence;
               end if;
               set table showing line is available;
               -- ready to transmit next message
          end if;
     end loop get next message;  -- forever

end send_mode_II_and_IV;
```

Figure 4. GOTO-Free PDL With Explicit State Variable Assignments

```
task body send_mode_ii_and_iv is

  ...

begin -- send_mode_ii_and_iv

  accept init (line: physical_line) do
    phys_line := line;
  end init;
  sf_sf := (line_tbl_ops.spec_term_type'( interswitch ) =
    line_tbl_ops.read_spec_term( phys_line ) );

<< startup >>
  notify_operator(( startup, phys_line ));

  loop   -- forever

<< between >>
    tasks.table( phys_line ).output.get_message( message );

<< som_seq >>
    buffer( 1..ti_lgth ) := gen_transmission_id(
      read_char_set( message ));

    for i in 1 .. ti_lgth
    loop
      trans.mit( buffer( i ));
    end loop;     -- for i in 1 .. ti_lgth

    decoup := new decoup_acc;
    decoup.log(( log_type => som_out,
      block   => 0,
      line    => phys_line,
      msg     => message,
      seq_num => line_tbl_ops.read_ocsn( phys_line ) ));
    decoup := null;

    open_for_read( message => message,
      start_with => line_tbl_ops.read_start_part( phys_line ),
      pos        => pos );

    buffer_length := char_count'last;
    status := ok;
```

Figure 5. Original Code

```
<< msg_seq >>
    while status = ok
    loop
      read_continuous( message => message,
        count      => buffer_length,
        where_from => pos,
        text       => buffer,
        status     => status );

      for i in 1..buffer_length
      loop
        select
          task.table( phys_line ).output.
            check_for_preempt( message );

          cantran_seq;

          decoup := new decoup_acc;
          decoup.log(( log_type => cantran_out,
           blocks  => 0,
            line    => phys_line,
            msg     => message,
            seq_num => line_tbl_ops.read_ocsn(
                   phys_line ) ));
          decoup := null;

          goto som_seq;

        else

          trans.mit( buffer( i ));

        end select;  -- check_for_preempt
      end loop;    -- for i in 1..buffer_length

      end loop; -- while status = ok
```

Figure 5. Original Code (Continued)

```
      if status /= end_of_text then

      cantran_seq;

      decoup := new decoup_acc;
      decoup.log(( log_type => cantran_out,
        blocks  => 0,
        line    => phys_line,
        msg     => message,
        seq_num => line_tbl_ops.read_ocsn( phys_line ) ));
      decoup := null;

      svc := new gen_svc;
      svc.generate_svc_message(( msg_ref => message,
        msg_type => suspended_transmission ));
      svc := null;

      tasks.table( phys_line ).output.finished_sending;
      notify_operator( (
        soft_hard_malfunction,
        "send_ii_and_iv",
        err_stat'image( status )
          & " encountered in read of message for
            transmission." ));

      goto between;

    end if; -- status /= end_of_text

<< eom_seq >>
    buffer( 1..eom_lgth ) := eomseq;

    for i in 1..eom_lgth
    loop
      trans.mit( buffer( i ));
    end loop; -- eom_seq      -- for i in 1..eom_lgth

    decoup := new decoup_acc;
    decoup.log(( log_type => eom_out,
      line    => phys_line,
      msg     => message,
      seq_num => line_tbl_ops.read_ocsn( phys_line ) ));
    decoup := null;

    increment_ocsn( phys_line );

<< trlr_seq >>
    if not sf_sf then
      buffer( 1..trlr_lgth ) := trailer( read_char_set ( message ));

      for i in 1..trlr_lgth
      loop
        trans.mit( buffer( i ));
      end loop; -- trlr_seq     -- for i in 1..trls_lgth

    end if; -- not sf_sf

    task.table( phys_line ).output.finished_sending;

  end loop;     -- forever

end send_mode_ii_and_iv;
```

Figure 5. Original Code (Continued)

```
task body send_mode_ii_and_iv is

  ...

begin -- send_mode_ii_and_iv

  accept init (line: physical_line) do
    phys_line := line;
  end init;
  sf_sf := (line_tbl_ops.spec_term_type'( interswitch ) =
    line_tbl_ops.read_spec_term( phys_line ) );

-- startup

  notify_operator(( startup, phys_line ));

  loop   -- forever

-- send message

    tasks.table( phys_line ).output.get_message( message );

    status := ok;    --  ready to send message

send message:
    while status = ok or status = preempted
    loop
       buffer( 1..ti_lgth ) := gen_transmission_id(
        read_char_set( message ));

      for i in 1 .. ti_lgth
      loop
        trans.mit( buffer( i ));
      end loop;     -- for i in 1 .. ti_lgth

      decoup := new decoup_acc;
      decoup.log(( log_type => som_out,
        block   => 0,
        line    => phys_line,
        msg     => message,
        seq_num => line_tbl_ops.read_ocsn( phys_line ) ));
      decoup := null;

      open_for_read( message => message,
        start_with => line_tbl_ops.read_start_part( phys_line ),
        pos        => pos );

      buffer_length := char_count'last;
```

Figure 6. Revised Code

```
send_characters:
     loop       -- exit at end when status /= ok
       read_continuous( message => message,
         count      => buffer_length,
         where_from => pos,
         text       => buffer,
         status     => status );

       for i in 1..buffer_length
       loop
         select
           task.table( phys_line ).output.
             check_for_preempt( message ); -- gets preempting message

           cantran_seq;

           decoup := new decoup_acc;
           decoup.log(( log_type => cantran_out,
             blocks  => 0,
             line    => phys_line,
             msg     => message,
             seq_mun => line_tbl_ops.read_ocsn(
                    phys_line ) ));
           decoup := null;

           status := preempted;

         else      -- send character

           trans.mit( buffer( i ));

         end select;  -- check_for_preempt
       end loop;    -- for i in 1..buffer_length

     end loop send_characters; -- while status = ok
   exit when status /= ok;
   end loop send_message;    -- exit if not preempted and
             -- (done sending or error
             -- condition)
```

Figure 6. Revised Code (Continued)

```
    if status /= end_of_text then

      cantran_seq;

      decoup := new decoup_acc;
      decoup.log(( log_type => cantran_out,
        blocks  => 0,
        line    => phys_line,
        msg     => message,
        seq_num => line_tbl_ops.read_ocsn( phys_line) ));
      decoup := null;

      svc := new gen_svc;
      svc.generate_svc_message(( msg_ref => message,
        msg_type => suspended_transmission ));
      svc := null;

      tasks.table( phys_line ).output.finished_sending;
      notify_operator( (
        soft_hard_malfunction,
        "send_ii_and_iv",
        err_stat'image( status )
          & " encountered in read of message for
            transmission." ));
    else     -- status = end_of_text

-- end of message sequence

      buffer( 1..eom_lgth ) := eomseq;

      for i in 1..eom_lgth
      loop
        trans.mit( buffer( i ));
      end loop; -- eom_seq     -- for i in 1..eom_lgth

      decoup := new decoup_acc;
      decoup.log(( log_type => eom_out,
        line    => phys_line,
        msg     => message,
        seq_num => line_tbl_ops.read_ocsn( phys_line ) ));
      decoup := null;

      increment_ocsn( phys_line );

-- send trailer

      if not sf_sf then
        buffer( 1..trlr_lgth ) := trailer( read_char_set (message ));

        for i in 1..trlr_lgth
        loop
          trans.mit( buffer( i ));
        end loop; -- trlr_seq    -- for i in 1..trlr_lgth

      end if; -- not sf_sf

      task.table( phys_line ).output.finished.sending;
    end if;

  end loop;    -- forever

end send_mode_ii_and_iv;
```

Figure 6. Revised Code (Continued)

THIS PAGE INTENTIONALLY LEFT BLANK

ARRAY OF ARRAYS

1. BACKGROUND

Case Study Objective

To illustrate how an array of arrays may be clearer and more appropriate than a multidimensional array.

Designer's Problem

What alternatives does the designer have in developing multi-dimensional array data structures?

Discussion

Ada, unlike many earlier languages, does allow arrays of arrays. The designer, however, may have extensive experience in these earlier languages and may thus be accustomed to using other constructs such as multidimensional arrays. An effective training program must take such backgrounds into account and must provide explanations and examples of Ada's novel features.

2. DETAILED EXAMPLE

Example Problem Statement

Radar returns (blips) from each sector are associated and form tracks. (The Case Study TASK STRUCTURE FOR A TARGET TRACKING SYSTEM explains this process in greater detail.) These tracks, in turn, must be stored in some kind of data structure, sorted by the sector to which they belong.

Solution Outline

An initial attempt to represent this data structure was as follows:

```
type SECTOR_DATA is array (TRACK_NO'FIRST .. TRACK_NO'LAST)
                          of TRACK_RECORD;
type TRACK_STORAGE_FILE is array (SECTOR'FIRST .. SECTOR'LAST,
                                  TRACK_NO'FIRST .. TRACK_NO'LAST)
                                of TRACK_RECORD;
TRACK_STORAGE: TRACK_STORAGE_FILE;
LOCAL_DATA : SECTOR_DATA;
```

This method states that any track record in the track storage file may be accessed by identifying the sector number and track number. This implies that the track storage file is a collection of track records, when what the designer sought to convey is that the track storage file is a collection of sector data.

With this data structure, transferring sector data for local processing requires a loop:

```
CUR_SECTOR := ...;
for I in TRACK_NO'FIRST .. TRACK_NO'FIRST + NUM_TRACKS -1
loop
     LOCAL_DATA(I) := TRACK_STORAGE(CUR_SECTOR,I);
end loop;
```

Ada allows the designer to declare a more natural data structure: an array of arrays. This approach is shown in the detailed solution.

Detailed Solution

```
type SECTOR_DATA is array (TRACK_NO) of TRACK_RECORD;
type TRACK_STORAGE_FILE is array (SECTOR) of SECTOR_DATA;

-- Note that in specifying the index, stating the type TRACK_NO or
-- SECTOR is sufficient and is equivalent to using the attributes
-- FIRST and LAST to specify the range, i.e., the range
-- TRACK_NO'FIRST .. TRACK_NO'LAST or SECTOR'FIRST .. SECTOR'LAST.
```

Transferring a set of sector data is now much more straightforward:

```
LOCAL_DATA := TRACK_STORAGE(CUR_SECTOR)
                    (TRACK_NO'FIRST..TRACK_NO'FIRST + NUM_TRACKS -1);
```

The loop in the previous example is now written as a slice of TRACK_STORAGE (CUR_SECTOR), which is an array of TRACK_RECORDS.

3. EPILOGUE

This exercise illustrates an usage for an array of arrays. Ultimately, it is track records that are the basic unit of information stored in the track storage file. The viewpoint, however, is different when this file is organized as a two-dimensional array of track records than when it is declared as a one dimensional array of sector data, which in turn an array of track records. The second approach, moreover, corresponds to the level of abstraction inherent in the designer's objective.

Section 6

GENERAL OBSERVATIONS AND CONCLUSIONS

6.1 Issues of Greatest Concern

The three areas of greatest concern that were noted during the technical interchanges with the contractors were tasks, packages, and exceptions. There exists a need for guidelines on what constitutes the appropriate use of these constructs. Regarding tasking, it was felt that the designer should assume an ideal situation in which tasking overhead and interprocessor switching did not impact performance. Proceeding from this assumption, the designer should then design the system using as many tasks as desired. Later, when compromises have to be made between the desired run-time system and the available run-time system, the designer has two choices. User code may be written to provide the performance and support needed for the current design, or those parts of the current design which are not possible with the available run-time system may be isolated and redesigned with a fall-back strategy for that kind of system.

6.2 Hardware Error Detection

If a given hardware component has built-in test equipment (BITE), how should errors be signalled to the software? At first, exceptions would probably seem to be the appropriate mechanism, however, that is not necessarily the case. The disadvantage of the exception mechanism is that the exception might be raised in the wrong task! The exception might be reasonable when the error is detected while performing a specific operation (for example, when accessing a device). When error detection is spontaneous and asynchronous with respect to the software, it would seem preferable to cause an interrupt, or to "post" an error indicator that can be interrogated by the software at opportune times.

6.3 Reusable Software Modules

The advantages of reusable modules are fundamental to the entire Ada concept and will not be discussed here. An issue that emerged during the study was how the Department of Defense would go about promoting the development of such modules. An example that was often mentioned concerns communication protocols. Using a common module is the best way to ensure that two systems use the same protocol!

With enough attention, every procurement could generate several off-the-shelf modules by suitably generalizing modules produced for that particular procurement. Should contractors be given extra incentives to produce reusable modules? Should there be an organization which monitors procurements and identifies potentially reusable modules? Should such

an organization recommend that the contractor be given extra incentives to generalize the modules, or should the reusable module be developed independently?

6.4 Customized Run-time Systems

Designers are often confused by the interface between the design and the run-time system. For example, the language does not specify the scheduling regime (round-robin? time-sliced?); similarly, distributed rendezvous may or may not be supported. Concerns also exist regarding the performance of the run-time system. How does one estimate what a "reasonable" number of tasks is? How does one estimate the time certain events will take?

The answer to this is that the run-time system is just another system component; eventually one will be able to buy one off-the-shelf, or have one custom-made, or have to live with one that is already available. The economic tradeoffs will change with time. The availability of off-the-shelf components will make the implementation of custom-built run-time systems both uneconomical and unnecessary. The question that is completely open (and whose answer will be determined by marketplace forces) is, when Ada reaches a "steady state," will there really be a rich choice of economical run-time systems? Or will we have to live with a small set of all-purpose implementations?

6.5 Need for Automated SDL and PDL Processing Tools

In using Ada as the SDL and PDL, the contractors found that it would be useful to have tools to check the correctness of their SDL and PDL. Furthermore, a need for tools which would maintain updated data dictionaries, data flow diagrams, structure charts, etc. were identified to obviate the need for manual updates to all these forms of system documentation whenever a change to one of these or the SDL or PDL were made. The SDL and PDL are not meant to be compilable, executable Ada code, thus a different kind of tool than a compiler would be needed to perform checks on its correctness. However, many of the tools needed are similar to those that might be provided with an Ada compiler (e.g. formatters, syntax-directed editors, cross-reference generators). One kind of tool that was considered highly desirable was one that would produce a graphical representation of a program structure from the SDL or PDL.

6.6 Versatility of Ada in the System Life Cycle

Both contractors noted that writing Ada code was greatly facilitated by the outputs of previous phases. Using Ada at the different phases of the system life cycle gave a measure of consistency to the outputs of these phases and enhanced the communication between them. However, because Ada is in fact a programming language, difficulties were noted in using Ada at the different phases: it was unclear where the boundary between requirements definition and system design, system design and

detailed design, and design and coding lay. The undefinable nature of the boundary is seen in questions such as how can the designer tell when sufficient decomposition has been done at a particular level? In using Ada for requirements definition and design, there is a danger of coding too soon, of worrying about low-level details at too early a stage, in short, of losing the forest for the trees. This problem is best addressed by establishment of appropriate guidelines and checks for each life cycle phase, and enforcement of these by project management. The value of using Ada throughout the life cycle outweighs the potential danger of having too much detail too soon. Program managers face the challenge of realizing these benefits while establishing procedures to avert potential problems.

APPENDIX A
AREAS FOR FUTURE RESEARCH

During the study, several issues were identified as needing further research, either because lack of time prevented a complete analysis, or because the issues arose as a "philosophical" question, and the design exercises did not provide enough material for a realistic case study. The issues can be grouped in two broad categories:

1. Unsettled questions. These are issues on which tentative guidelines were identified, but insufficient experience exists.

2. Promising ideas. In many cases a design problem was identified as a classic for which it would be interesting to develop a "cookbook" of known approaches, with guidelines for selection.

A.1 REQUIREMENTS DEFINITION PHASE

A.1.1 Expressing High-Level Performance Constraints

This area involves developing ways in Ada of formulating nonfunctional requirements, in particular performance constraints on the system. Ada does not have a specific construct which may be written at the beginning of a procedure to state that this operation must not take longer than say, 5 milliseconds, and that it must perform with a very high mean time between failures. As another example, Ada does not offer a formal mechanism to state the requirement that the system must use a particular military protocol and interface with certain equipment. A promising idea is to complement the functional requirements with a model of relevant external agents. This idea deserves further study. (Reference Sections 2.2, 2.4)

A.1.2 Single-threaded Protocols

The issue here is how to specify the processing required for a logical stimulus independent of low-level concerns and multi-threaded requirements. A classical mistake is to combine these concerns, producing an obscure low-level specification. This leads to confusion at all subsequent levels of the system life cycle. What needs to be done is to develop specification and design paradigms, and investigate whether the designer should be required contractually to design the system using single-threaded protocols. (Reference Sections 2.2, 2.3)

A.2 PRODUCT DESIGN PHASE

A.2.1 Steady State Approach to Real Time Systems

This topic is concerned with segregating steady state issues from mode transition issues (e.g., error handling) in specifying the design of a system. A problem in designing a real time system occurs in having to design simultaneously both the system's normal operation and the procedures to follow in the event of a malfunction. From a requirements point of view, an error recovery function is independent of the steady state function. From a design point of view, however, error recovery is quite difficult in that it affects many different steady-state processes. The steady state approach facilitates the identification and specification of the major functions. With the control issues being addressed separately, the design is easy to manage and the designer can specify exactly the requirements for the support modules at the next stage of design. The idea seems promising but has not been adequately tested. (Reference Section 3.2)

A.2.2 Methodological Issue: Multiple/Distributed Processing

This issue addresses the methodological requirements the user faces during the design process of a system to be implemented with multiple or distributed processing. In this area Ada offers considerable expressive power, but it is uncertain to what extent various implementations will support such power. Examples of implementation problems include task allocation between processors, the meaning of an access type when it is passed from one processor to another, the implications of an interprocessor rendezvous, the effect of calling a procedure which is stored and runs on a different processor than the currently active process, and exception propagation between processors.

Ideally, the designer should not worry about these issues and assume that the appropriate interprocessor communications exists. In the implementation phase, the system developers should then write the low-level customized routines to support the needed interprocessor communications. This research area focuses on two issues: the handling of distributed processing in Ada in current design methodologies, and the definition and analysis of the interprocessor communications issues. (Reference Section 3.2)

A.2.3 Guidelines in Designing One Task per "What"

In developing a tasking model for a system, the designer must choose what object or activity will be represented by a task. For instance in a radar application, one task could be allocated per blip (or stimulus), or one task could be allocated per sector scanned (or sensor). The system design will be significantly different for a task per stimulus than for a task per sensor approach. "Cookbook" solutions using both approaches should be developed, with guidelines for selecting between them. (Reference Sections 3.3, 3.6)

A.2.4 Information Hiding - Data Abstraction

In the course of system design it is often desirable to specify data structures and operations on these structures without concerning oneself with the details of their implementation. This allows the designer to address design issues in clearly defined levels of abstraction and usually results in a clean design. The package feature of Ada supports the desired separation of concept and implementation, thus encouraging information hiding in the design process. Further paradigms and guidelines are needed on specifying packages to encapsulate the levels of abstraction in an abstract data type and its available operations. (Reference Section 3.4)

A.2.5 Generic Types: System Security and Testability

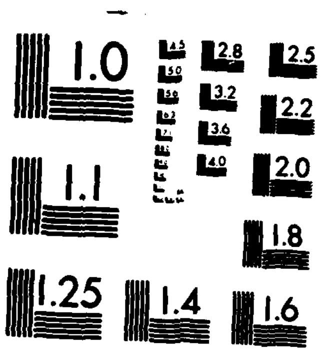
The use of private types as generic parameters provides a possible approach for keeping critical system parameters hidden. If certain parameters (e.g., maximum number of targets to be tracked) are specified as generic parameters, then the system can be exercised with specific values for testing purposes without revealing the actual values needed by the system. After testing is concluded, instantiation with the real values will yield a system with the desired capacity. The extent to which this approach will help to address the classified nature of some software needs further study. (Reference Sections 3.4, 4.3)

A.2.6 Use of Parameters at a Particular Level of Design

Both functions and data are identified in the design phase, and the designer must determine the extent to which the data should be parameterized. While the operations on the data objects are specified in the functional subprograms designed, the question arises as to whether the design is the appropriate place to specify what data is global and what objects are passed as parameters. At the top levels of design, subprograms were not parameterized, thereby deferring the decisions of what to parameterize until a later stage of the design. This approach makes sense in that it enables the designer to get a complete view of the data in the system, both where and at what level of abstraction it is used. With this global understanding of the data objects, the designer can analyze them to identify potential packages. The issue to be studied here is at what level of design to introduce parameters into entry and subprogram calls. (Reference Section 3.4)

A.2.7 Managing a Common Storage Pool

The requirements for a system may state that certain actions must be taken when a percentage of dynamic storage has been allocated. The language does not provide a facility for monitoring storage directly or for showing what data is stored in a particular data heap (or zone with a data heap). Although low-level routines can be written to provide this kind of support, system modularity is compromised. Research must be done to determine the optimum solution to this problem. (Reference Section 3.5)



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

A.2.8 Modeling a Finite State Machine

The abstract notion of a finite state machine is based on the machine transiting from one state to another given a particular input value. The properties of Ada tasking provide an elegant model of an automaton using selective waits and the blocking property of the Ada rendezvous. There are several ways to implement state transitions. The use of Ada tasks in modeling the state transitions of a finite state automaton is an area which needs to be further explored. (Reference Section 3.7)

A.2.9 Alternative Task Selection

Depending on a condition one of a given set of tasks may be allocated. This condition, however, is not known until run-time and it is only evaluated once at run-time. The problem that arises is that any time that an entry in one of the potentially allocated tasks is to be called the condition must be evaluated again. This adversely affects the clarity and readability of the code. By reversing the direction of the call (i.e. the allocated task does the calling, as opposed to its entry being called) this problem can be avoided. Further research is needed on the impact of the direction of calls in the Ada rendezvous. (Reference Section 3.8)

A.2.10 Postponing Design Details

Ada provides a number of features which enable the designer to postpone design details until a clear and coherent picture of the system being designed has emerged. These features include generics, private types, procedures, stubbing, parameterization, package specification, etc. For example, if a designer is unsure as to how to model a particular activity, that activity can be designated (and invoked) as a procedure. The advantage of an Ada procedure is that it provides the designer with significant latitude at the later stages of system development. (The design of this activity may result in, to name two of several possibilities, tasks embedded within this procedure, or an entry call into another task.) Similarly, at the top level of design, a subprogram may not be parameterized in order to defer a decision on what data should be global and what should be local. More research is needed to determine the circumstances under which these features should be used by the designer as well as the extent of their use. (Reference Sections 3.4, 3.9, 5.6 [DECOUPLING PARTLY INDEPENDENT ACTIVITIES], A.2.6)

A.3 DETAILED DESIGN PHASE

A.3.1 Modeling Tasks with SADT, DFD's, Structure Charts

Traditional methods of graphically specifying system structure do not provide a notation for expressing Ada tasking. Research is needed into how tasking can be expressed in a manner consistent with commonly used graphical notation. (Reference Section 4.2)

A.3.2 Guidelines on Packages

A set of criteria should be developed to aid the designer in deciding which program units and data definitions belong in which package. For example, issues such as information hiding, object-oriented design, commonality of use of groups of user-defined entities (such as types, data, operations, subprograms, etc.), the designer's intuition, modularity, size need to be explored further as they relate to packages. (Reference Sections 2.4, 3.4, 4.3, 5.2)

A.3.3 Tutorial on Exceptions

Exceptions have some of the characteristics of global data (and, in fact, may be considered a form of side effect). They are undoubtedly a powerful design tool, but they are easy to misuse. A methodology for their proper usage needs to be developed. (Reference Sections 4.4, 5.5)

A.4 CODE/UNIT TEST PHASE

A.4.1 Dot Notation, Naming and Readability

Each programmer tends to develop his or her own coding style. For ease of reading the code and maintaining it, some agreement on a standard coding style is needed. Naming conventions should be adopted as well. Dot notation, while it has the advantage of identifying the parent package(s) of a piece of data or a subprogram, does result in more cumbersome code when too many parents are identified. One possibility is to rename all lengthy parent names at the beginning of subprograms. Another alternative is to develop a tool which will take a program and supply a new version of this program using dot notation. Thus, a programmer could use simpler names, knowing that these names could automatically be expanded to show their source. Explicit guidelines for the use of dot notation, use clauses and renaming need to be developed.. (Reference Section 5.2)

A.4.2 Standard Coding Style

To promote readable and maintainable code, some effort should be directed to defining a standard coding style. Such a style would likely address matters of formatting, naming conventions, comments, program structure (e.g., nested packages vs. library units), stubbing, module size, etc. (Reference Section 5.2)

A.4.3 Avoiding Busy Waits in Select Statements

Depending on the direction of call between two tasks (task A calls an entry in task B or task B calls an entry in task A) the following scenarios may exist. First, task A is waiting for task B to finish some activity before they rendezvous, and task A executes a busy wait loop. In other words, it is in a loop which selects either to accept the entry call from B or to do nothing (i.e., null;). Second, task A must again wait for task B to complete some computation, but instead of waiting for

B to wake it up, it calls the entry in B. As soon as B is ready, it accepts the call and the rendezvous may proceed. A case study exemplifying these situations and exploring the reasons why busy waits should be avoided would be an instructive case study. (Reference Section 4.5 [TASK PREEMPTION])

A.4.4 Simple Coding Paradigms

During the study, a sizable number of simple examples of particularly Ada-like usage were observed but could not be properly written up as case studies, due to lack of time. An example is the use of a character type in a case statement. It would be worthwhile to capture these simple examples.

A.4.5 Handling Impossible States

This issues addresses handling unexpected or presumably impossible system states. The impossible condition is a well known phenomenon in every software project. The typical example is a case statement in which some of the cases are known to be impossible in a certain context. A different situation arises when the specification is unclear about some of the cases. A serious difficulty was perceived to be that of educating designers and programmers to be aware of the problem. At the beginning of each project the technical mechanisms should be established, but most of all there should be a standard practice whereby a programmer or module designer can promptly call attention to an impossible situation and receive guidance for its handling. Research is needed to identify such procedures and to develop programming guidelines so that coders do not overlook or deliberately hide impossible states. (Reference Sections 4.4, 5.5)

A.5 GENERAL ISSUES

A.5.1 Hardware Error Detection

This issues address how a hardware component's built-in test equipment should signal errors to the software. Research is needed to determine when it is reasonable to raise an exception, when it is preferable to cause an interrupt and when an error indicator should be set which may then be checked by the software. (Reference Section 6.2)

A.5.2 Reusable Software Modules

This topic involves identifying mechanisms to promote the development of reusable software modules. For example, when two systems must communicate using the same protocol, the best way to ensure this is to use a common module or package. With enough attention, every procurement could generate several off-the-shelf modules by suitably generalizing modules produced under that contract. Further study is needed to find possible ways of identifying and generating reusable software. (Reference Section 6.3)

A.5.3 Customized Run-Time Systems

This topic is concerned with the specification of the interfaces between the system and the run-time support environment. The run-time system is another system component, and eventually it will be available off-the-shelf. Depending on the application, a customized run-time system may be needed which has, say, a super-efficient implementation to support tasking and distributed rendezvous. If such a run-time system is not available, the designer of the system could either build an interface to the run-time system to support distributed rendezvous or modify the design so that this feature is no longer required. There are several areas here which need further study: the economic and design trade-offs between customized and general-purpose run-time systems, the extent to which a type of run-time system should be standardized, and the types of interfaces needed in a run-time system to facilitate system design and development. (Reference Section 6.4)

A.5.4 Parameters to Analyze Tasking Performance

In order to evaluate aspects of a particular system design, parameters should be identified to analyze tasking performance. Examples of such parameters are the type of memory, the speed of the processor, the size of the program, and the time for task switching. The relationship between these factors needs to be studied in order to develop a measure of tasking performance. Also to be examined is the methodological question, at which stage or stages of the design should the performance parameters be taken into account? (Reference Section 6.4, 6.6)

A.5.5 Analysis of User Requirements

A recurring problem, on which there was unanimous agreement, is that user's specifications are invariably incomplete, obscure and inconsistent. There was a feeling that writing precise specifications is the designer's task, and that a specific feedback mechanism should be incorporated into any realistic methodology, to the point of being an explicit contractual requirement. This idea is far-reaching and worthy of further thought. (Reference Sections 2.2, 2.3, 2.4, 3.2, 3.5, 4.4, 5.5)

APPENDIX B

TABLE OF ADA LANGUAGE FEATURES PRIMARILY ADDRESSED IN CASE STUDIES

Ada Language Features \ Case Studies	Power Failure Requirements	Use of Types to Describe Hardware Interface Requirements	Functional Description of an Air Defense System	Task Structure for a Target Tracking System	UART: Expressing Hardware Design in Ada	Tasks and Structure Charts	Use of Dependent Tasks	Task Preemption	Queues and Generics	Stubbing and Readability	Succinctness of Range Syntax	Rendezvous and EXIT	Decoupling Partly Independent Activities	Memory-Mapped I/O in Ada	Eliminating GOTO's	Array of Arrays
Lexical Rules					X					X	X		X		X	
Character Set																
Lexical Units											X		X		X	
Identifiers										X					X	
Numeric Literals																
Character Literals																
String Literals																
Comments															X	
Pragmas																
Reserved Words																
Allowed Replacement of Characters																
Declaration and Types (General Concepts)	X	X		X	X			X	X	X	X		X	X	X	X
Declarations		X			X									X		
Objects and Named Numbers	X	X			X					X			X			
Types and Subtypes		X														
Type Declarations		X														
Subtype Declarations		X														
Derived Types		X														
Scalar Types		X								X						
Enumeration Types		X			X					X						
Character Types																
Boolean Types								X							X	
Integer Types				X							X					
Real Types				X												
Array Types		X		X	X					X						X
Index Constraints and Discrete Ranges		X		X						X	X					
The Type String																
Record Types		X		X						X	X					
Variant Parts									X					X		
Access Types										X			X			
Declarative Parts		X								X						
Names and Expressions (General Concepts)	X	X			X					X	X	X	X	X	X	X
Names					X					X		X				
Slices																X

Ada Language Features \ Case Studies	Power Failure Requirements	Use of Types to Describe Hardware Interface Requirements	Functional Description of an Air Defense System	Task Structure for a Target Tracking System	UART: Expressing Hardware Design in Ada	Tasks and Structure Charts	Use of Dependent Tasks	Task Preemption	Queues and Generics	Stubbing and Readability	Succintness of Range Syntax	Rendezvous and EXIT	Decoupling Partly Independent Activities	Memory-Mapped I/O in Ada	Eliminating GOTO's	Array of Arrays
Attributes										X						X
Literals																
Aggregates		X									X					
Expressions										X	X			X	X	
Operators and Expression Evaluation											X				X	
Type Conversions	X															
Qualified Expressions																
Allocators													X			
Static Expressions and Static Subtypes																
Universal Expression																
Statements (General Concepts)		X			X			X		X		X		X	X	X
Simple and Compound Statements - Sequence of Statements																
Assignment Statement										X				X	X	
Array Assignments		X			X											X
If Statements											X	X			X	
Case Statements																
Loop Statements								X							X	X
Block Statements					X											
Exit Statements								X		X		X			X	
Return Statements																
Goto Statements															X	
Subprograms (General Concepts)	X	X			X	X	X	X		X			X	X		
Subprogram Declarations		X				X	X			X						
Formal Parameter Modes		X				X				X						
Subprogram Bodies						X	X	X		X						
Subprogram Calls						X	X	X		X						
Function Subprograms						X		X		X						
Parameter and Result Type Profile - Overloading of Subprograms																
Overloading of Operators																
Packages (General Concepts)	X	X			X	X			X	X						
Package Structure		X			X	X				X						
Package Specifications and Declarations	X	X			X				X							

Ada Language Features / Case Studies	Power Failure Requirements	Use of Types to Describe Hardware Interface Requirements	Functional Description of an Air Defense System	Task Structure for a Target Tracking System	UART: Expressing Hardware Design in Ada	Tasks and Structure Charts	Use of Dependent Tasks	Task Preemption	Queues and Generics	Stubbing and Readability	Succinctness of Range Syntax	Rendezvous and EXIT	Decoupling Partly Independent Activities	Memory-Mapped I/O in Ada	Eliminating GOTO's	Array of Arrays
Package Bodies	X				X				X							
Private Types and Deferred Constant Declarations		X														
Limited Types						X										
Visibility Rules (General Concepts)	X	X			X				X	X						
Scope of Declarations										X						
Use Clauses	X	X			X				X							
Renaming Declarations	X									X						
Tasks (General Concepts)	X			X	X	X	X	X	X	X		X	X	X		
Task Specifications and Task Bodies	X			X	X	X	X	X	X			X	X			
Task Types and Task Objects				X			X	X					X			
Task Execution - Task Activation	X			X	X			X				X	X			
Task Dependence - Termination of Tasks							X	X					X			
Entries, Entry Calls and Accept Statements	X			X	X	X	X	X	X	X		X	X			
Delay Statements, Duration and Time	X				X			X								
Select Statements	X			X	X		X	X	X			X	X			
Selective Waits	X			X				X								
Conditional Entry Calls								X								
Priorities	X															
Task and Entry Attributes																
Abort Statements								X								
Shared Variables														X		
Program Structure and Compilation Issues (General Concepts)	X	X		X		X	X			X				X		
Compilation Units - Library Units	X			X		X	X			X				X		
Context Clauses - With Clauses		X				X										
Subunits of Compilation Units										X						
The Program Library							X							X		
Program Optimization														X		

Ada Language Features \ Case Studies	Power Failure Requirements	Use of Types to Describe Hardware Interface Requirements	Functional Description of an Air Defense System	Task Structure for a Target Tracking System	UART: Expressing Hardware Design in Ada	Tasks and Structure Charts	Use of Dependent Tasks	Task Preemption	Queues and Generics	Stubbing and Readability	Succintness of Range Syntax	Rendezvous and EXIT	Decoupling Partly Independent Activities	Memory-Mapped I/O in Ada	Eliminating GOTO's	Array of Arrays
Exceptions (General Concepts)					X		X		X							
Exception Declarations					X		X									
Exception Handlers					X		X		X							
Raise Statements					X		X									
Exception Handling							X									
Exceptions Raised During Task Communication							X									
Suppressing Checks																
Generic Units (General Concepts)		X			X	X	X		X					X		
Generic Declarations		X			X	X	X		X							
Generic Bodies					X				X							
Generic Instantiation		X			X				X					X		
Representation Clauses and Implementation Dependent Features (General Concepts)	X													X		
Representation Clauses														X		
Interrupts	X													X		
Change of Representation																
Machine Code Insertions																
Interface to Other Languages																
Unchecked Programming and Type Conversion																
Input-Output (General Concept)						X								X		
General User-Level Input-Output						X										
Text Input-Output																
Low-Level Input-Output														X		

LME

— 8